AMD

AMD SimNow™ Simulator 4.5.0

User's Manual

Revision	Date
2.10	**April 2009**

Advanced Micro Devices, Inc.
One AMD Place
Sunnyvale, CA 94088

simnow@amd.com

© 2004-2009 Advanced Micro Devices, Inc.

The Contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising by estoppels or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Trademarks

AMD, the AMD Arrow logo, AMD Athlon, AMD Opteron and combinations thereof, SimNow, 3DNow!, AMD-8111, AMD-8131, AMD-8132 and AMD-8151 are trademarks of Advanced Micro Devices, Inc.
HyperTransport is a trademark of the HyperTransport Technology Consortium.
Microsoft and Windows are registered trademarks of Microsoft Corporation.
PCI-X is a registered trademark of PCI-SIG.
Sysmark is a registered trademark of Business Applications Performance Corp.
MMX is a trademark of Intel Corporation.
Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Contents

Figures ix
Tables xi

1 Overview 1
2 Installation 3
2.1 System Requirements 3
2.2 Installation Procedure 3
2.3 Directory Structure and Executable 4
2.4 Setting up Linux for the Simulator 4
2.5 Configuration File 5
2.6 Updates and Questions 6
3 Graphical User Interface 7
3.1 Tool Bar Buttons 7
3.2 Device Window 9
3.2.1 Add a New Device 10
3.2.2 Workspace Popup Menu 10
3.2.2.1 Add Connection 11
3.2.2.2 Configure Device 12
3.2.2.3 Disconnect Device 12
3.2.2.4 Delete Device 13
3.2.3 Example Computer Description 13
3.2.4 Device Window – Quick Reference 15
3.3 Device Groups 15
3.3.1 Terms 16
3.3.2 Concept Diagrams 17
3.3.3 Working with Device Groups 18
3.3.4 Shell Automation Commands for Device Groups 19
3.3.4.1 Device Tree 19
3.3.4.2 Enabled vs. Disabled vs. Mixed 20
3.3.5 Device Group Examples 21
3.3.5.1 Example: 1GB DDR2 memory 21
3.3.5.2 Example: Quad-Core Node 22
3.3.5.3 Example: SuperIO device 24
3.3.6 Creating a Device Group (GUI) 24
3.3.7 Creating a Device Group (Automation Commands) 27
3.3.8 Ungrouping a created device group 29
3.4 Main Window 29
3.4.1 SimStats and Diagnostic Ports 29
3.4.2 CPU-Statistics Graphs 30
3.4.2.1 Translation Graph 30
3.4.2.2 Real MIPS Graph 30
3.4.2.3 Invalidation Rate Graph 31
3.4.2.4 Exception Rate Graph 31

3.4.2.5 PIO Rate Graph 31
3.4.2.6 MMIO Rate Graph 32
3.4.3 Simulated Video 32
3.4.4 Hard Disk and Floppy Display 32
3.4.5 Using Hard Drive, DVD-/CD-ROM and Floppy Images 33
3.4.6 Help, Problems and Bug Reports 33
4 Disk Images 35
4.1 Creating A Blank Hard-Drive Image 35
5 Running the Simulator 39
5.1 Command-Line Arguments 39
5.1.1 Open a Simulation Definition File 40
5.2 Installing an Operating System 42
5.2.1 Assigning Disk-Images 42
5.2.2 Run the Simulation 44
5.2.3 Interaction with the Simulated Machine 45
5.2.4 Simulation Reset 45
5.3 Multi-Machine Support 45
6 Create a Simulated Computer 49
6.1 BSD Files 49
6.2 Device Placement 49
6.3 Solo.bsd Device Configuration 51
6.4 Save and Run 52
7 Device Configuration 53
7.1 AweSim Processor Device 55
7.2 Debugger Device 58
7.3 DIMM Device 59
7.4 Emerald Graphics Device 65
7.5 Matrox MGA-G400 PCI/AGP 69
7.6 Super IO Devices: Winbond W83627HF SIO / ITE 8712 SIO 78
7.7 Memory Device 81
7.8 PCA9548 SMB Device 84
7.9 PCA9556 SMB Device 85
7.10 AMD 8th Generation Integrated Northbridge Device 86
7.11 AMD-8111™ Southbridge Devices – IO Hubs 90
7.12 PCI BUS Device 96
7.13 AMD-8131™ PCI-X® Controller 98
7.14 AMD-8132™ PCI-X® Controller 99
7.15 PCI-X Test Device 101
7.16 AMD-8151™ AGP Bridge Device 102
7.17 Raid Device: Compaq SmartArray 5304 104
7.18 SMB Hub Device 105
7.19 AT24C Device 107
7.20 EXDI Server Device 108
7.21 USB Keyboard and USB Mouse Devices 109
7.22 XTR Device 110
7.22.1 Using XTR 111

7.22.1.1 Recoding XTR Trace 111
7.22.1.2 Stop XTR Record 111
7.22.1.3 XTR Playback 111
7.22.1.4 Stop XTR Playback 112
7.22.2 XTR Structure 114
7.22.2.1 XML Structure 114
7.22.2.2 XTR Binary File Contents 116
7.22.3 ModeFlags 116
7.22.4 Limitations 117
7.22.5 Example XTR XML File 117
7.23 JumpDrive Device 123
7.24 E1000 Network Adapter Device 124
7.24.1 Simulated Link Negotiation 125
7.24.2 The Mediator Daemon 126
7.24.3 MAC Addresses for use with the Adapter 127
7.24.4 Example Configurations 127
7.24.4.1 Absolute NIC 127
7.24.4.2 Client-Server simulated network 128
7.24.4.3 Isolated Client-Server simulated network (Same process) 128
7.24.5 Visibility Diagram 129
7.25 Plug and Play Monitor Device 130
7.26 ATI SB400/SB600/SB700 Southbridge Devices 132
7.27 ATI RS480/RS780/RD790/RD890 Northbridge Devices 134
7.28 AMD “Istanbul” Device 135
7.29 AMD “Sao Paulo” Device 136
7.30 AMD “Magny-Cours” Device 137
7.31 AMD “DeerHound” Device 138
8 PCI Configuration Viewer 139
8.1 Scanning PCI Buses 140
8.2 Modifying the PCI Configuration contents 140
9 Logging 141
9.1 Message Log 141
9.2 Error Log 143
9.3 I/O Logging 144
10 CPU Debugger 147
10.1 Using the CPU Debugger 147
10.1.1 Setting a Breakpoint 147
10.1.2 Single Stepping the Simulation 148
10.1.3 Stepping Over an Instruction 148
10.1.4 Skipping an Instruction 149
10.1.5 Viewing a Memory Region 149
10.1.6 Reading PCI Configuration Registers 150
10.1.7 Reading CPU MSR Contents 150
10.1.8 Find Pattern in Memory 151
10.2 Debugger Command Reference 151
11 Debug Interface 155

11.1 Kernel Debugger 155
11.2 GDB Interface 156
11.2.1 Simple Approach 156
11.2.2 Alternate Approach 157
11.2.3 Using Another Port on the Same Machine 157
11.2.4 Using Two Separate Machines 157
11.3 Linux Host Serial Port Communication 157
12 Command API 159
13 DiskTool 161
13.1 Command-Line Mode 161
13.2 GUI Mode 162
14 BIOS Developer's Quick Start Guide 167
14.1 Loading a BIOS Image 167
14.2 Changing DRAM Size 167
14.3 Changing SPD Data 168
14.4 Clearing CMOS 169
14.5 Logging PCI Configuration Cycles 169
14.6 Logging CPU Cycles 170
14.7 Creating a Floppy-Disk Image 171
15 Frequently Asked Questions (FAQ) 173
A Appendix 177
A.1 Format of Floppy and Hard-Drive Images 177
A.2 Bill of Material 178
A.2.1 Computer Platform Files (BSD) 178
A.2.2 Device Files (*.BSL) 178
A.2.3 Product Files (*.ID) 178
A.2.4 Image Files (*.HDD, *.FDD, *.ROM, *.SPD, *.BIN) 179
A.2.4.1 Hard-Disk Image Files 179
A.2.4.2 Memory SPD Files 180
A.3 Supported Guest Operating Systems 181
A.4 CPUID 182
A.4.1 CPUID Standard Feature Support (Standard Function 0x01) 182
A.4.2 CPUID AMD Feature Support (Extended Function 0x80000001) 182
A.5 Known Issues 184
A.5.1 FSAVE/FRSTOR and FSTENV/FLDENV 184
A.5.2 Triple Faulting 184
A.5.3 Performance-Monitoring Counter Extensions 184
A.5.4 Microcode Patching 184
A.5.5 Instruction Coherency 184
A.6 Instruction Reference 186
A.6.1 Notation 186
A.6.1.1 Mnemonic Syntax 186
A.6.1.2 Opcode Syntax 188
A.6.2 General Purpose Instructions 189
A.6.3 System Instructions 217
A.6.3.1 INT – Interrupt to Vector 219

A.6.3.2 IRET – Return from Interrupt 219
A.6.4 Virtualization Instruction Reference 220
A.6.5 64-Bit Media Instruction Reference 220
A.6.6 3DNow!™ Instruction Set 220
A.6.7 Extension to the 3DNow! Instruction Set 221
A.6.8 Prescott New Instructions 221
A.6.8.1 MONITOR – Setup Monitor Address 222
A.6.8.2 MWAIT – Monitor Wait 223
A.7 Automation Commands 224
A.7.1 Shell 225
A.7.2 IDE 229
A.7.3 USB 230
A.7.4 CMOS 231
A.7.5 ACPI 231
A.7.6 Floppy 231
A.7.7 Debug 231
A.7.8 AMD-8151™ AGP Bridge 232
A.7.9 VGA 232
A.7.10 Serial 232
A.7.11 HyperTransport™ Technology Configuration 234
A.7.12 8th Generation Northbridge 235
A.7.13 DBC 235
A.7.14 AMD-8111™ Device 235
A.7.15 EHC 236
A.7.16 Journal 236
A.7.17 CPU 236
A.7.17.1 Profiling in SimNow™ Technology 236
A.7.17.2 CPU Code Generator Commands 238
A.7.18 Emerald Graphics 238
A.7.19 Matrox MGA-G400 Graphics 239
A.7.20 PCI Bus 239
A.7.21 SIO 239
A.7.22 Memory Device 240
A.7.23 Raid 241
A.7.24 DIMM 242
A.7.25 Keyboard and Mouse 243
A.7.26 JumpDrive 244
A.7.27 E1000 247
A.7.28 XTR 247
A.7.29 ATI SB400/SB600/SB700 248
A.7.30 ATI RS480 248
A.7.31 ATI RS780 249
A.7.32 ATI RD790/RD780/RX780 249
A.7.33 ATI RD890S/RD890/RD780S/RX880 249

Index 251

Figures

Figure 3-1: Main Window (In Simulation) 7
Figure 3-2: Device Window 9
Figure 3-3: Workspace Popup Menu 11
Figure 3-4: Add Connection Dialog of Device Properties Window 12
Figure 3-5: Computer Simulation in "cheetah_1p.bsd" File 13
Figure 3-6: Device group: BSD with one machine group and three child devices 17
Figure 3-7: Device group (different conceptual view – devices are inside groups) 18
Figure 3-8: Device Group (2 group devices 1 library device) 18
Figure 3-9: Modify Group 18
Figure 3-10: Device Group 19
Figure 3-11: Example DIMM Device Group 21
Figure 3-12: Created DIMM Device Group 22
Figure 3-13: Children of DIMM Device Group 22
Figure 3-14: Group Devices 24
Figure 3-15: Console Window 29
Figure 3-16: Progress Meter and Diagnostic Ports 30
Figure 3-17: CPU Translation Graph 30
Figure 3-18: CPU Real MIPS Graph 31
Figure 3-19: CPU Invalidation Graph 31
Figure 3-20: CPU Exception Rate Graph 31
Figure 3-21: CPU PIO Rate Graph 32
Figure 3-22: CPU MMIO Rate Graph 32
Figure 3-23: Primary, Secondary, and Floppy Displays 33
Figure 4-1: DiskTool Dialogue Window 36
Figure 4-2: DiskTool Shell Window 36
Figure 4-3: New Image Size 37
Figure 4-4: Create Blank Image 37
Figure 4-5: DiskTool Operation Successful 38
Figure 5-1: Main Window (No BSD Loaded) 39
Figure 5-2: Main Window (BSD Loaded) 41
Figure 5-3: Device Window 42
Figure 5-4: Installing WindowsXP 44
Figure 5-5: Special Keys Generator 45
Figure 6-1: Solo.bsd Configuration 49
Figure 6-2: Connections Tab of Device Properties Window 50
Figure 6-3: PCI Bus Configuration dialog window 52
Figure 7-1: AweSim Processor-Type Properties 56
Figure 7-2: AweSim Processor Logging Properties Dialog 57
Figure 7-3: AMD Opteron™ Processor Virtual Bank-Select Line Configuration 60
Figure 7-4: AMD Athlon™ 64 Processor Bank-Select Line Configuration 60
Figure 7-5: DIMM-Bank Options Properties Dialog 62
Figure 7-6: DIMM Module Properties Dialog 63
Figure 7-7: Graphics-Device VGA Sub Device Properties Dialog 66

Figure 7-8: Graphics Device Frame Buffer SubDevice Properties 67
Figure 7-9: Matrox G400 Block Diagram 69
Figure 7-10: Matrox G400 Information Property Dialog 71
Figure 7-11: Matrox G400 Configuration Properties 72
Figure 7-12: Enable Full Hardware Acceleration on WindowsXP guest 77
Figure 7-13: Super IO Properties Dialog: Winbond W83627HF 79
Figure 7-14: Memory Configuration Properties Dialog 82
Figure 7-15: PCA9548 SMB Configuration Properties Dialog 84
Figure 7-16: PCA9556 SMB Configuration Properties Dialog 85
Figure 7-17: Northbridge Logging Capabilities Properties Dialog 87
Figure 7-18: Northbridge HT Link Configuration Properties Dialog 87
Figure 7-19: Northbridge DDR2 Training Properties Dialog 88
Figure 7-20: USB Properties Dialog (AMD-8111™ Southbridge) 91
Figure 7-21: CMOS Properties Dialog (AMD-8111™ Southbridge) 92
Figure 7-22: HDD Primary Channel Properties Dialog (AMD-8111 Southbridge) 93
Figure 7-23: Device Options Properties Dialog (AMD-8111 chipset) 94
Figure 7-24: Logging Options Properties Dialog (AMD-8111 chipset) 95
Figure 7-25: PCI Bus Properties Dialog 97
Figure 7-26: AMD-8131™ Device Hot Plug Configuration 98
Figure 7-27: AMD-8132™ Device Hot Plug Configuration 99
Figure 7-28: AMD-8132 Properties Dialog 100
Figure 7-29: AMD-8151™ Device Properties Dialog 102
Figure 7-30: SMB Hub Properties Dialog 106
Figure 7-31: AT24C Device Configuration 107
Figure 7-32: Communication via Mediator 124
Figure 7-33: Multi-Machine Communication without a Mediator 125
Figure 7-34: Visibility Diagram 129
Figure 7-35: Plug and Play Monitor Device Configuration 131
Figure 7-36: ATI SB600 SATA Configuration Dialog 133
Figure 8-1: PCI Configuration Viewer 139
Figure 9-1: Message Log 142
Figure 9-2: Error Log 143
Figure 9-3: I/O Logging Dialog 144
Figure 10-1: CPU Debugger Window 147
Figure 13-1: DiskTool Shell Window 163
Figure 13-2: DiskTool GUI Window 164
Figure 13-3: DiskTool Drive Information 164
Figure 13-4: DiskTool Progress Window 165
Figure 14-1: Memory Configurator 168
Figure 14-2: Diagnostics Display 169
Figure 14-3: Message Log Window 170
Figure 15-1: Console Window 224

Tables

Table 1-1: Feature Overview Public Release versus Full Release 2
Table 2-1: Software and Hardware Requirements 3
Table 3-1: Cheetah_1p.bsd Devices 15
Table 3-2: Device Window - Quick Reference 15
Table 3-3: Image Types 33
Table 5-1: Command-Line Arguments 40
Table 5-2: Newmachine Command Arguments 46
Table 7-1: Supported Devices 54
Table 7-2: Supported Standard VESA Modes 67
Table 7-3: Supported Custom VESA Modes 68
Table 7-4: Matrox G400 VESA Modes 75
Table 7-5: Supported Resolutions in Power Graphics Mode 75
Table 7-6: Supported Guest Operating Systems 76
Table 7-7: Execution Control Flags 116
Table 7-8: Internal Execution Control Flags 117
Table 7-9: Mediator Command Line Switches 127
Table 7-10: MAC Address Assignments 128
Table 7-11: Client-Server: Simulator Server 128
Table 7-12: Client-Server: Simulator Client 1 128
Table 7-13: Isolated Client-Server: Simulator Server 128
Table 7-14: Isolated Client-Server: Simulator Client 1 129
Table 10-1: Debugger Breakpoint Command Examples 148
Table 10-2: Debugger Memory Dump Command Examples 150
Table 10-3: Debugger Pacifica Memory Dump Command Examples 150
Table 10-4: MSR Read Examples 151
Table 10-5: MSR Write Example 151
Table 10-6: Find Pattern Example 151
Table 10-7: Debugger Commands and Definitions 154
Table 15-1: Computer Platform Files (BSD) 178
Table 15-2: Product Files 179
Table 15-3: Hard-Disk Images 180
Table 15-4: Memory SPD Files 180
Table 15-5: Supported Guest Operating Systems 181
Table 15-6: CPUID Standard Feature implementation 182
Table 15-7: CPUID Extended Feature implementation 183
Table 15-8: General-Purpose Instruction Reference 217
Table 15-9: System Instruction Reference 219
Table 15-10: 3DNow!™ Instruction Reference 221
Table 15-11: Extension to 3DNow! Instruction Reference 221
Table 15-12: Prescott New Instruction Reference 222
Table 15-13: CodeGen Command Overview 238
Table 15-14: Prefix Sequences (keyboard.text) 244

1 Overview

The AMD SimNow™ simulator is an AMD64 technology-compatible x86 platform simulator for AMD's family of processors. It is designed to provide an accurate model of a computer system from the program, OS, and programmer's point of view. It allows fast simulation of an entire computer system, plus standard debugging features such as break-pointing, memory-viewing, and single-stepping. The simulator allows such work as BIOS and OS development, memory-parameter tuning, and multi-processor system simulation.

Section 2.1, "*System Requirements*", on page 3 describes supported host Operating Systems. Section A.3, "*Supported Guest Operating Systems*", on page 180 describes supported guest Operating Systems.

The simulator has between a 10:1 and 100:1 slowdown rate from the host CPU, depending on whether the workload is in the CPU core or accessing simulated devices intensively.

The simulator is designed to create an accurate model of a system from the program's view. Device models contain all the program-visible state but the actual functionality is abstracted. In many cases only the functionality needed to satisfy the software is implemented. Software may be run on the simulator in an unmodified form. This includes BIOS, drivers, O/S, and applications.

The simulator has a concept of time, but it is not a cycle-accurate simulator. The basic timing mechanism is an instruction; all instructions execute in the same amount of time and are one tick in length. This "tick" time is scaled and used by the rest of the system. Long-latency events, like disk or floppy access, have some minimum latency built in because we found legacy software that relied on the physical latency of these peripherals.

The simulator contains all the classic pieces of a PC system (CPU, memory, Northbridge, Southbridge, display, IDE drives, floppy, keyboard, and mouse support). Images (hard disk, DVD/CD-ROM, and floppy) can be created in custom sizes with the DiskTool program (Section 13, "DiskTool", on page 161) that is provided with the simulator. A simulation can be saved at any point in the simulation to a media file, from which the simulation can be re-run at a later time.

A simple diagnostic port model (known as "Port80" device) displays values written by the BIOS in a pane of the simulator's main window. Other panes display guest (simulated machine) and simulator host processor times. The simulator requires several files to be specified. Binary files containing the BIOS and disk images are stored in the images directory. The simulator home directory stores "**.bsd*" files which contain the configuration of the system (how models are connected together and their settings) and the logical state of all the devices in the simulator. When starting a simulation from reset, the "**.bsd*" file is rather small and only contains the configuration information. When the simulation starts, the simulated memory is allocated. When the simulation is halted and

saved, the "**.bsd*" file will have grown significantly, slightly larger than the size of simulated memory.

The graphics device supplied with the simulator is a 2D and 3D graphics card with linear frame buffer and DirectX 6 support. AMD currently plans to provide a graphics model with the simulator which will also have modern 3D hardware acceleration, including Microsoft® DirectX 9/10 support.

The simulator is available in two versions: *Public Release* and *Full Release*. Table 1-1 shows the detailed feature matrix:

Feature	Public Release	Full Release
DIMM configuration	*Limited*	✔
No 4 Gb limitation of simulated memory	✘	✔
Available devices	*Limited*	✔
Available platform definition files (BSDs)	*Limited*	✔
Devices can be added and removed from platform definition files	✔	✔
Connecting and disconnecting devices	✔	✔
Ships with a variety of different CPU cores (Product Files)	✔	✔
Full product support	*Limited*	✔
Analyzer support	✔	✔
Support of simulated multi-processor systems (up to 16 CPUs)	✔[1]	✔

Table 1-1: Feature Overview Public Release versus Full Release

To get more information about how to obtain the full release version of the simulator please send an email to simnow@amd.com.

[1] Support of up to two cores.

2 Installation

2.1 System Requirements

The AMD SimNow™ simulator runs on both Linux 64 for AMD systems and Windows® for 64-bit AMD systems.

The requirements for each system are as follows:

	Linux 64 for AMD64	**Windows® XP 64Bit Edition for AMD64**
OS Distribution *Recommended*	Any of the following 64-Bit Linux distributions for AMD64. • SuSE 9 Pro and newer • RedHat 64Bit Enterprise 3 and above • Fedora Core 2 and newer. SuSE 9.1 or newer for AMD64.	Windows XP x64 Edition or Windows Server 2003 x64 Edition for AMD64. Build 1218 or newer.
Memory	Approx. 64MB of memory, plus Approx. 150 MB of memory for each simulated processor, plus the amount of simulated RAM.	
Processor	AMD Athlon™ 64 or AMD Opteron™.	
Hard Disk Space	1 Gigabyte of free hard disk space for the simulator and devices plus 3 Gigabytes free space for disk file images.	
Other Hardware	3.5-inch, 1.44-MB floppy drive. CD-ROM Drive.	

Table 2-1: Software and Hardware Requirements

Running the simulator on a Linux kernel prior to version 2.6.10 may cause the simulator to malfunction. The bug is in the 64-bit path only, and the symptom is in signals that are not associated with "system calls" still being treated as "system calls" as they go back to user space, i.e. in certain cases it tries to restart the "system call" even when it did not come from a "system call". Updating the Linux kernel to kernel version 2.6.10 or later resolves this problem.

The simulator may stress the system more than most applications, including the base operating system. AMD has received reports that the simulator has caused some systems to crash, and in general this has been traced to unstable hardware. Hardware instability can also crash applications or operating systems inside the simulator.

2.2 Installation Procedure

Insert the CD-ROM into your system's CD-ROM drive, or download the simulator program and its data files from http://developer.amd.com/simnow.aspx. Browse to the root directory of the CD or to the path where the downloaded simulator is stored, and

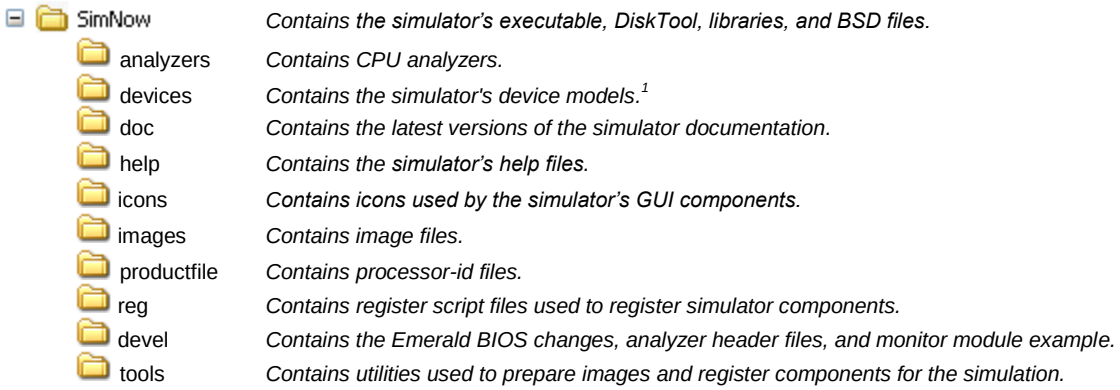
begin the installation, as follows. To install under Windows, double-click on the self-extracting executable. To install under Linux, extract the zipped tar file as shown below:

```
tar –xzf Simnow-Linux64-<version>.tar.gz
```

2.3 Directory Structure and Executable

After the opening screen and license agreement are displayed, you will be prompted to choose an installation directory. When you select this, the install program will copy the executable files and device models to the selected directory and setup the registry entries necessary to run the simulator.

The install program will create the following subdirectories under the install directory:


SimNow
Contains the simulator's executable, DiskTool, libraries, and BSD files.
analyzers
Contains CPU analyzers.
devices
Contains the simulator's device models.[1]
doc
Contains the latest versions of the simulator documentation.
help
Contains the simulator's help files.
icons
Contains icons used by the simulator's GUI components.
images
Contains image files.
productfile
Contains processor-id files.
reg
Contains register script files used to register simulator components.
devel
Contains the Emerald BIOS changes, analyzer header files, and monitor module example.
tools
Contains utilities used to prepare images and register components for the simulation.

[1] *Under Windows each model is a Windows DLL. Under Linux each model is a Linux library. Each model has a ".bsl" extension.*

2.4 Setting up Linux for the Simulator

Make a file: "*/etc/sysctl.conf*" (or add to the existing one)

```
# This is here to make sure we get enough "mmap"able virtual address
# space, in 4K pages.  It defaults to 65536, which is generally
# too small.
vm.max_map_count = 1048576

# This line doesn't need to be here for newer Linux kernels, but some
# early AMD64 Linux kernels would log SEGVs even if a process had a
# handler for them, which is what SimNow does.
debug.exception-trace = 0
```

Example 2-1: Setting up Linux for the Simulator

Then run "*sysctl -p*", or make sure the boot sequence does this if you don't want to run it at each reboot.

Newer Linux distributions may set a per-process memory limit by default. SimNow allocates a large amount of memory that is never touched. This untouched memory will not be backed by DRAM or swap, but Linux counts it against SimNows process memory limit when it comes to resource limits.

You can unset the per-process memory limits by running the following commands as root.

```
ulimit -m unlimited
ulimit -v unlimited
```

2.5 Configuration File

The simulator's configuration file is a text file that may be edited and that is stored in different locations depending on which host OS you are using.

If you are using Windows as host operating system the configuration file is located in:

```
C:\Documents and Settings\All Users\Application Data\simnowrc
```

If you are using Linux as host operating system the configuration file is located in:

```
$HOME/.qt/simnowrc
```

Here is an example of the contents of this file, with an explanation:

```
[General]

[UserKeys]
CTL-ESC=Sends a CTL-ESC to the application,1D 01 81 9D
ALT-F4=Sends an ALT-F4 to the application,38 3e be b8

[UserBottons]
BUTTON0=”MyIconPath\MyIcon.png”,“cpu.name”
```

The configuration file is divided into sections, with each section title enclosed in square brackets. This particular example includes three sections, named [General], [UserKeys] and [UserBottons].

All user key definitions are stored in the [UserKeys] section. Each user key definition is defined by a single line. This example defines two user keys. The string to the left of the equal sign is the string that will be placed in the menu. To the right of the equal sign are two strings, separated by a comma. The first string is the text that is displayed when the user clicks on the "*What's This*" help button, and the second string is the list of scan codes that are sent when this menu item is selected.

The two examples shown can also be generated by the “*Generate Key Codes*” menu item on the “*Special Keyboard*” menu, see Section *5.2.3, “Interaction with the Simulated Machine”,* on page *45*.

All user button definitions are stored in the [UserButtons] section. Each user button definition is defined by a single line. This example defines one user button (BUTTON0). The string to the left of the equal sign is the path including the file name of the icon that will be placed in the toolbar menu. To the right of the equal sign is the string that

represents the automation command (please refer to Section A.7, "*Automation Commands*", on page 224) that will be executes when the user clicks on the defined user button.

Note that minimal parsing of the text is done, so it is important that **no spaces** exist around the separating comma.

2.6 Updates and Questions

Please refer to the *Release Notes* located at "*SimNow\docs*" to obtain the latest information about the simulator. If you have any question regarding the simulator please refer to Section 15, "*Frequently Asked Questions (FAQ)",* on page 173 or contact your AMD account representative.

Appendixes are provided that describe:

- Format of Floppy and Hard-Drive Images, page 177
- Bill of Material, page 178
- Supported Guest Operating Systems, page 180
- CPUID, page 182
- Known Issues, page 184
- Instruction Reference, page 186
- Automation Commands, page 224

3 Graphical User Interface

The simulator has a cross-platform GUI that uses the Qt toolkit. We welcome bug reports and usability feedback on the simulator.

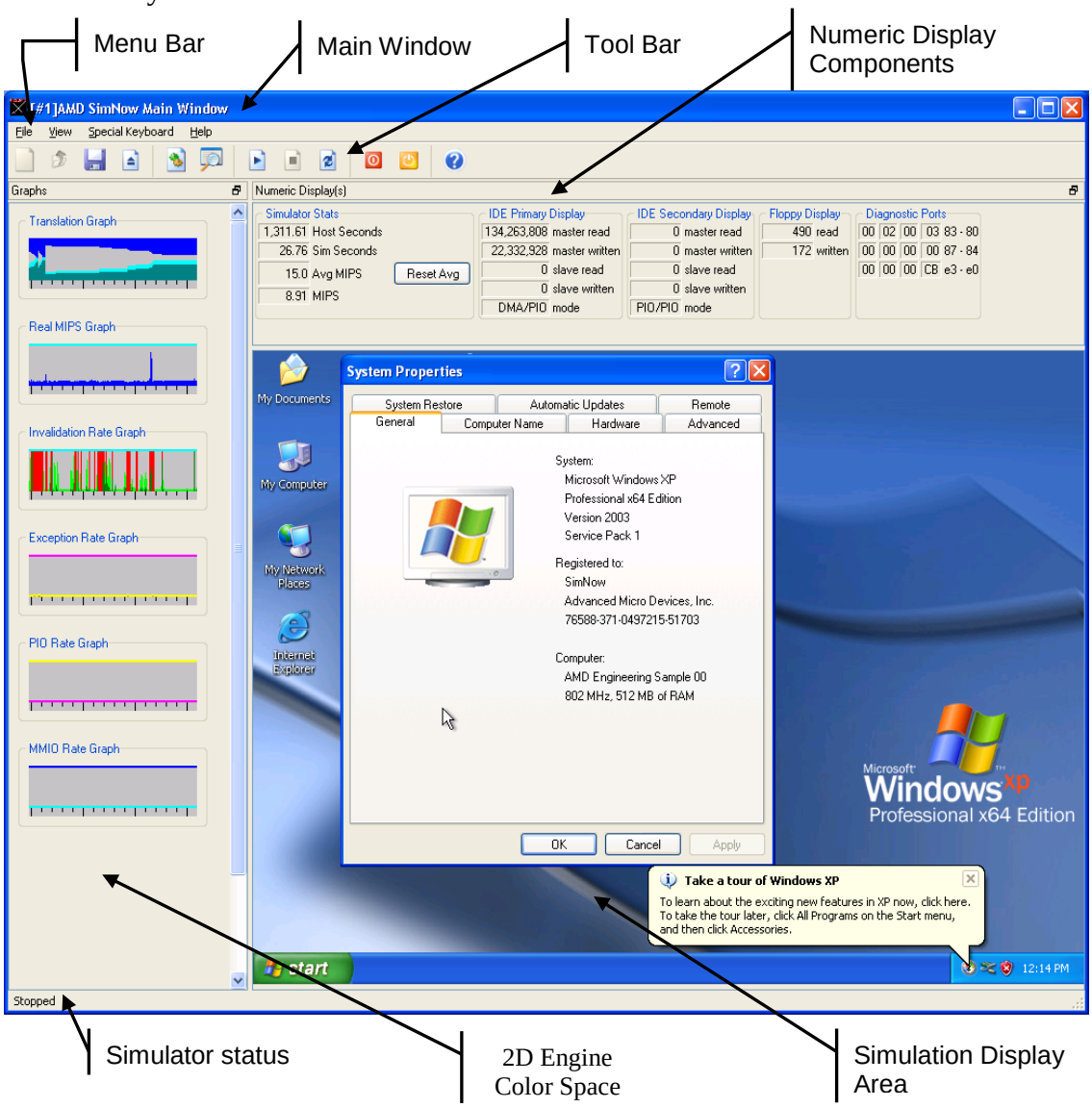


Figure 3-1: Main Window (In Simulation)

3.1 Tool Bar Buttons

The Tool Bar shown in Figure 3-1 contains up to eleven control buttons.

The simulation can be started by clicking on the "*Play*" button ( ).

The simulation can be stopped by clicking on the "*Stop*" button ( ). To reset the entire simulator, stop the simulation first by clicking on the "*Stop*" button and then click on the "*Reset*" button ( ).

The power-management "*Soft Power*" button ( ) and "*Soft Sleep*" button ( ) are available only on simulated systems that have an Advanced Configuration and Power Interface (ACPI) BIOS.

Clicking on the "*Soft Power*" button puts the simulated system in a very low power consumption mode. The working context can be restored if it is stored on nonvolatile media. The simulated system appears to be off.

Clicking on the "*Soft Sleep*" button simulates a power-consumption reduction. The power consumption is reduced to one of several levels, depending on how the system is to be used. The lower the level of power consumption, the more time it takes the system to return to the working state.

To close a previously loaded system simulation definition file click on the "*Close BSD*" button ( ). This button is only enabled when a system definition file has been loaded or created earlier. Please make sure you save any changes that you make to the system configuration before clicking on the "*Close BSD*" button ( ) to close the system definition file. Otherwise all changes will be lost.

The "*Save BSD*" button ( ) is only enabled/active when a system definition (BSD file) has been loaded or created. To save your current system definition click on the "*Save BSD*" button ( ) or click on the "*File*" menu item and select "*Save BSD*".

To open a system definition file (BSD file) click on the "*Open BSD*" button ( ) and select the desired BSD file from the Open File Dialog Window. The "*Open BSD*" button is only enabled/active when no other system definition file has been open yet.

To create a blank or new system definition file click on the "*New BSD*" button ( ). This button is disabled when a system definition file has been loaded or created earlier. In this case you must close your current system definition file, click on the "*Close BSD*" button ( ) or click on the "*File*" menu item and select "*Close BSD*". Please make sure you save any changes that have been made to the system definition file before you click on the "*Close BSD*" button ( ). Note, when closing the BSD file all changes will be lost.

If you want to modify the current system definition use the "*Show Device Window*" button ( ) to display the current system configuration. The "*Show Device Window*" button is disabled when the simulation is currently running. To stop the simulation click on the "*Stop Simulation*" button ( ).

To open the simulator's integrated debugger click on the "*Show Debugger*" button ( ). The "*Show Debugger*" button is disabled when the simulation is currently running. To stop the simulation click on the "*Stop Simulation*" button ( ).

Click on the "*Best Fit To Window*" button ( ) to reduce or enlarge the size of the simulated display area so that the entire simulated display area will fit into the simulators main window. If you hold down the CTRL key when clicking on the "*best fit*" button, it "locks" into a state where the simulated display area is resized whenever the simulated graphics display resolution changes. To clear this locked condition, click on the "*best fit*" button again.

3.2 Device Window

The Devices Window, shown in Figure 3-2, is opened by selecting "*View→Show Devices*" or clicking on the button. In this window, you can create a simulated computer and modify its properties, BIOS images, memory characteristics, and attached components.

This section describes the main components of the Device Window and shows how to build up and configure a simulated computer. It explains the interface using some of the most-often used simulation components. Please also see the walkthrough of building a single-processor system in Section 6, "*Create a Simulated Computer*", on page 49.

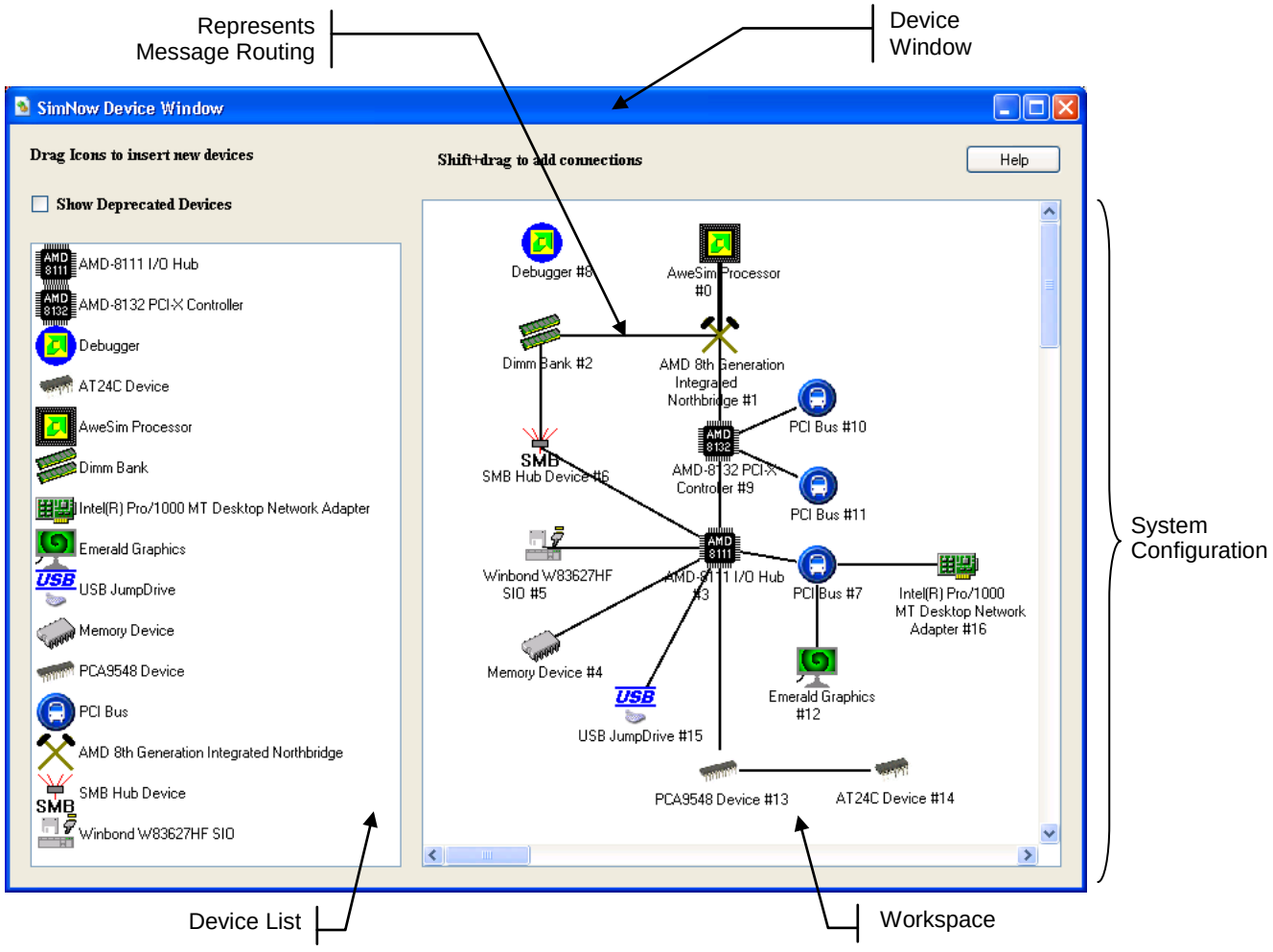


Figure 3-2: Device Window

The Device Window, shown in Figure 3-2 with the "*cheetah1_p.bsd*" computer simulation loaded, graphically depicts a simulated computer system. In the simulator, a computer system is defined as a collection of device models that communicate with each other by exchanging messages. The icons in the workspace represent device models; the lines connecting the icons represent message routing. You can set up and alter the simulated computer system by using the workspace popup menu (shown in Figure 3-3). To open the workspace popup menu, right-click on any icon in the workspace area.

The Device List, located on the left side of the Device Window, describes all devices available in the simulator along with their configuration options. For further information please refer to Section 7, "*Device Configuration*", on page 53.

The *Show Deprecated Devices* checkbox is not checked by default. This checkbox gives the user the opportunity to show or hide deprecated devices. It is not recommended to use deprecated devices in simulation. To show deprecated devices this checkbox must be checked. The *Show Deprecated Devices* checkbox does not disable the ability to connect or create deprecated devices. Also the automation interface of deprecated devices and loading BSDs which contain deprecated devices are both unaffected.

3.2.1 Add a New Device

You can add devices to the workspace by dragging a new device from the Device List on the left side of the workspace window to an appropriate location within the workspace on the right side.

Some devices produce additional windows or dialogs when you add them to the workspace. These windows provide an interface to the device during simulation. For example, adding the Winbond WB83627HF SIO device (see Section 7.5 on page 69) to the workspace adds the floppy byte counts numeric window to the Main Window screen.

When you add a device to the workspace, the shell sends a reset message to all of the devices in the workspace. The global reset is equivalent to power-cycling the simulated computer system.

3.2.2 Workspace Popup Menu

Changing the system configuration of the simulated system can make the simulation nonfunctional.

Right-clicking on any icon in the workspace produces a popup menu as shown in Figure 3-3.

Figure 3-3: Workspace Popup Menu

3.2.2.1 Add Connection

You can connect a device to another device by holding Shift, left-click, and drag from one device to the other. You will draw a line from the first device to the second. Release the mouse button to create the connection. You can also right-click one device, select "*Add Connection*", and then click on the device to connect to. Then click Finish. The connection enables simulator-level message exchanges between the connected devices. All connections enable bidirectional message transfers.

Some devices contain more than one interface to which a connection can be made. A multi-interface device routes messages out different interfaces, based on the type of message being sent. When you make a connection with a multi-interface device, an interface list dialog appears which enables you to select the appropriate interface. You must choose an interface on either device, even if one or both of the devices has only one interface type.

Generally, you shouldn't connect different types of interfaces. For example, interface *Type A* of Device 1 should only be connected to interface *Type A* of Device 2.

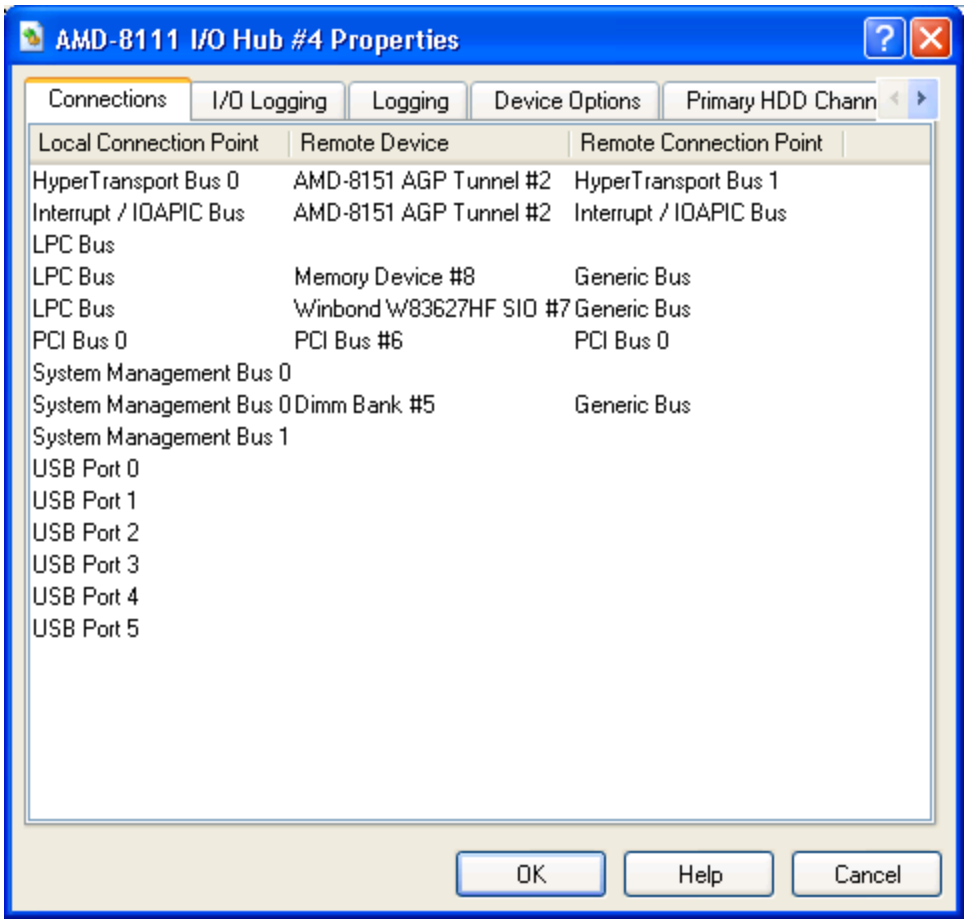


Figure 3-4: Add Connection Dialog of Device Properties Window

A device's connection appears in the "*Connections*" tab of the Device Properties window for each device, as shown in Figure 3-4.

When you add a connection, the simulator shell sends a reset message to all of the devices in the workspace. The global reset is equivalent to power-cycling the simulated computer system.

3.2.2.2 Configure Device

Most devices provide configuration options. Selecting "*Configure Device*" from the workspace popup menu produces a dialog window containing options for the specified device.

Selecting the "*Connections*" tab in the Device Properties window will display a list of all connections between the specified device and any other devices in the workspace.

3.2.2.3 Disconnect Device

Selecting "*Disconnect Device*" from the workspace popup menu removes all connections to the specified device.

3.2.2.4 Delete Device

Selecting Delete Device from the workspace popup menu removes all connections to the specified device, and removes the device from the workspace.

3.2.3 Example Computer Description

In this section we describe the major components of the computer simulation contained in the "*cheetah_1p.bsd*" file.

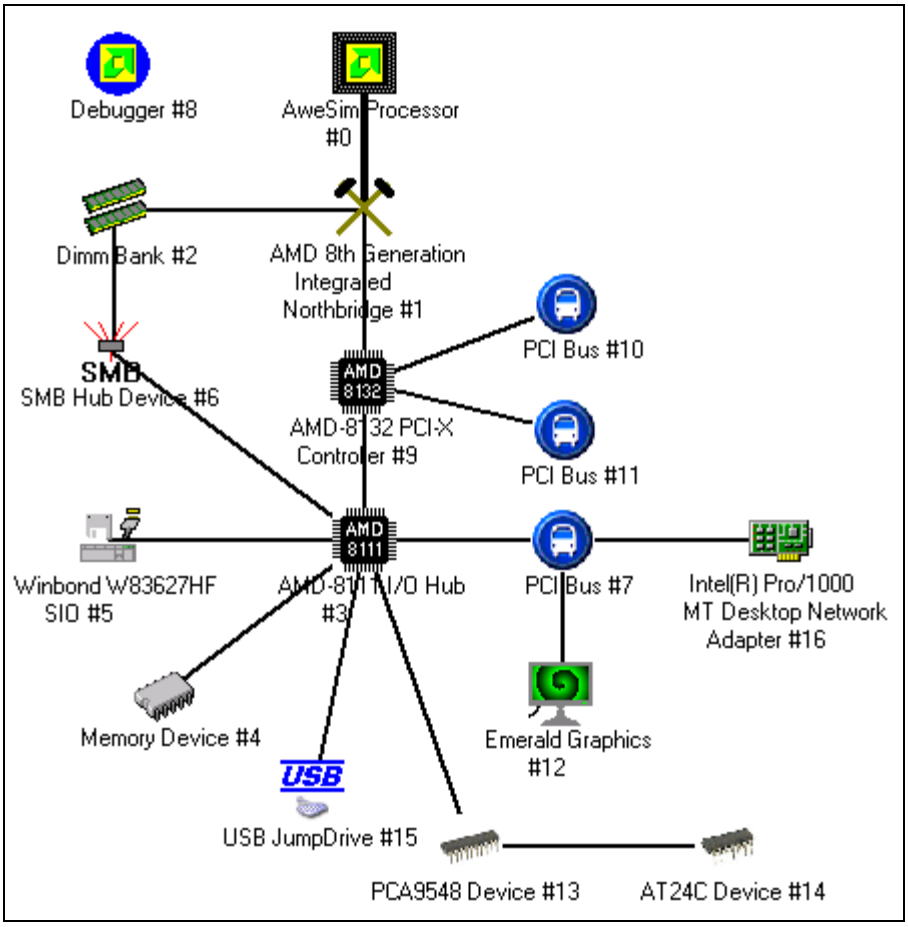


Figure 3-5: Computer Simulation in "cheetah_1p.bsd" File

This computer is a single-processor AMD 8th Generation machine with 256 MB of memory, a Southbridge that supports two IDE chains, VGA output, and a SuperIO that supports a keyboard, mouse, and floppy drive. This computer also comes with a USB JumpDrive and NIC device.

Right-clicking on any icon brings up a Workspace Popup menu (Figure 3-3) that allows access to the Device Property window, which includes a list of all components that the selected component is connected to. An example Device Property window is shown in Figure 3-4. The right-click Workspace Popup menu also allows you to delete or disconnect the selected device from all its connections.

Table 3-1 lists each component in the "*cheetah_1p.bsd*" computer. For more information about devices and possible device configuration, please refer to Section 7, "*Device Configuration*" on page 53.

Symbol	Device	Short Description
	AMD Debugger	Standard debugging support.
	AweSim Processor	Simulated Processor.
	DIMM Bank	DIMM Memory Modules.
	AMD 8th Generation Integrated Northbridge	Integrated Northbridge treated as a separate device in simulation.
	AMD-8111™ Southbridge	Southbridge supporting Hard drives, DVD-/CD-ROM drives, Floppy drives, USB ports, CMOS, and POST ports.
	AMD-8132™ PCI-X Controller	The AMD-8132 PCI-X Controller is a HyperTransport tunnel that provides two PCI-X buses and two IOAPICs. These PCI-X buses may or may not be configured as hot-plug-capable, depending on the platform.
	Emerald Graphics Device	Simulated VGA device.
	Matrox G400 Graphics Device	Simulated VGA/SVGA device.
	PCI Bus	Simulated PCI Bus which can connect multiple PCI devices (such as bridges and PCI VGA).
	Winbond W83627HF SIO	SuperIO Chip with keyboard, mouse and floppy.
	Memory Device	Device that contains a configurable BIOS ROM image.
	SMB Hub Device	The SMB hub device is used to connect one SMBus to any of four SMBus branches.
	PCA9548 Device	The PCA9548 is an 8-channel System Management Bus (SMB) switch.
	AT24C Device	The AT24C device is a Serial EEPROM device.
	USB JumpDrive	The JumpDrive device allows easy import and export of data between a host system and a simulation environment.

Symbol	Device	Short Description
	Desktop Network Adapter	The network adapter device models an Intel Pro/1000 MT Desktop Network Adapter.

Table 3-1: Cheetah_1p.bsd Devices

3.2.4 Device Window – Quick Reference

Table 3-2 lists common tasks that may be done in the Device Window and describes how to complete them.

Task	Where to Find the Properties
Change CPU Type	Enter the "*AweSim properties page→Processor*" tab and select a CPU type. For more information, please refer to Section 7.1, "*AweSim Processor Device, Figure 7-1* ", on page 56.
Change Memory type or size	Please refer to Section 14.2, "*Changing DRAM Size*", on page 167.
Change a hard drive or DVD-/CD-ROM image	Go to the *Simulation Display Window "File→Set IDE {Primary, Secondary} {Master, Slave} Image*", as shown in Figure 7-22, on page 93, ***Or*** Go to the "*Southbridge Properties page→HDD {Primary, Secondary} Channel*". If using a DVD-/CD-ROM image, check the DVD-ROM checkbox, as shown in Figure 7-22, on page 93.
Change a floppy drive image	Go to the Main Window "*File Menu→Set Floppy Image*" ***Or*** Go to the "*SIO properties page→Super IO*" *tab (see* Figure 7-13 on page 79*).*
Change a BIOS image	Go to the "*System BIOS Properties page→Memory Configuration*" tab (see Figure 7-14, on page 82). Change the Init File entry.

Table 3-2: Device Window - Quick Reference

3.3 Device Groups

A platform (*.bsd) consists of devices, and each device is an instance of either a device library (*.bsl or *.so) or a device group (*.bsg). A device group is an aggregation of devices into a single composite device that has some customized aspects (includes its name, icon, ports, initial and default state).

Device groups are a particular class of devices. They have the same properties and characteristics as traditional devices, but also allow the user to extend and tailor specific device(s) to meet a particular hardware implementation or configuration. Device groups provide a method that allows the user to *group* or *collect* one or more devices, libraries or groups into one composite device. To the user, the composite device will look and feel no

different than a *normal* device library and, for the most part, the two should be indistinguishable.

A device group can consist of one or more child devices, with some optional initialization state associated with each child device, and those devices can optionally be connected to each other. It may be helpful to think of a device group as a BSD within a BSD. However, a device group also has its own identity as a device, and it can support external connection ports that allow it be connected to other devices in the same manner as a traditional device library.

3.3.1 Terms

If any of the language and wording used in these Device Groups sections is unclear, it may help to refer to this list of terms.

Device: A device library or device group (also, a known device or created device).

Device Library: Contains binary implementation of device functionality; has no child devices; associated with a "*.bsl" Windows or "*.bsl" Linux file.

Device Group: Grouping of one or more devices (libraries and groups) into a single device; gets its functionality through aggregation of its children, and from its group-specific properties/aspects; associated with a "*.bsg" file.

Known Device: A device that the shell knows about (i.e., the shell has all the necessary information to create an instance of this device). Known devices appear in the left hand pane of the Device Viewer window; and on the console using shell.KnownDevices.

Created Device: An instantiation of a known device. All devices in a BSD are *created devices*. Created devices appear in the right hand pane of the *Device Viewer* window; and on the console using "`shell.CreatedDevices`".

Device grouping tree node relationships: Because of device grouping, created devices in a BSD are nodes in a tree, with parents and children, siblings, and end/root tree node relationships.

Device connection relationships: Because of device connections, a sibling device can be connected to another sibling device at a connection port of each device.

Machine Device Group: Just a device group, but it is special since it is the root node of a machine tree (it has no parent, it can't be deleted, it has no ports, and it has no sibling devices); each machine in a BSD has a single machine created device group.

Archive Data or Device State: A known device group has archive data for its child devices, which specifies the default and initial state for when a known device group is instantiated as a created device. A known device library also has default and initial state for when it is instantiated as a created device. When a BSD is saved, each device's current

state (archive data) (which may be different than the original known device's archive data) is saved to the "*.bsd" file.

Internal Connection: A connection between two children of a device group

External Connection: A connection between a device's parent group and a sibling of the parent group. Under-the-hood, a connection to a device group is routed to one of its children, via an internal-to-external port mapping between the child device's port and the parent device's port.

3.3.2 Concept Diagrams

A device group is a device with its own identity (name, description, icon, help file, etc). But it is also like a BSD; in fact, every BSD has a single created device group called the *Machine* device. Tthe default user's view into SimNow is from the context of looking inside the *Machine* device. This encapsulation of devices inside device group's results in a hierarchy tree, with a *Machine* device group as the root node. In this way, a device group tree is like a folder/directory tree (folder is to device group as file is to device library), as demonstrated in Figure 3-6.

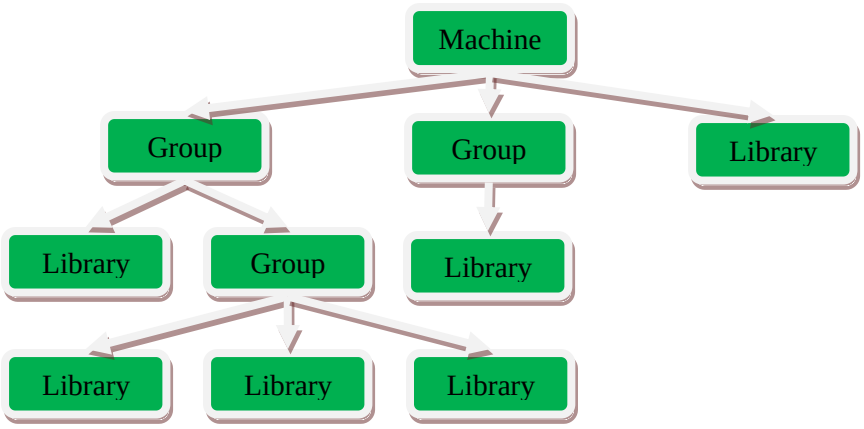


Figure 3-6: Device group: BSD with one machine group and three child devices

Any device can also connect to its sibling devices (Figure 3-6 does not depict any port connections). Figure 3-7 depicts the same example device tree, but with a different conceptual view - devices are inside groups; arrows represent possible port connections between sibling devices:

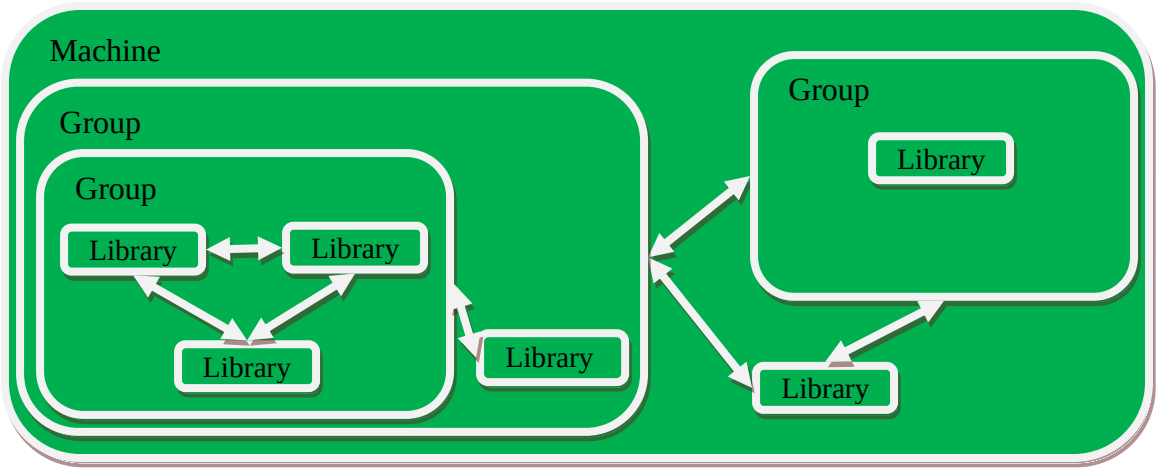


Figure 3-7: Device group (different conceptual view – devices are inside groups)

The previous diagrams show child devices inside device groups. On the standard top level view (the context of inside the machine device), we would more simply just see three devices, see Figure 3-8 (arrows represent possible port connections between the devices).

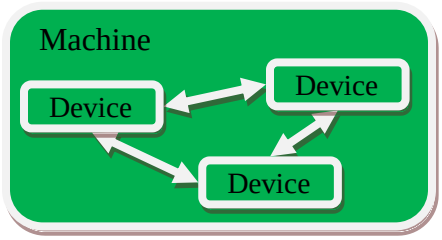


Figure 3-8: Device Group (2 group devices 1 library device)

3.3.3 Working with Device Groups

From the main SimNow window, "*View→Show Devices*", opens a device viewer GUI window for the machine device group. We can also open a device viewer GUI window that views any device group's children. Right-click the device icon and select "*Modify Group (Show Devices)*" from the popup menu. If "*Modify Group (Show Devices)*" is not present, then the device the user has clicked on is not a group.

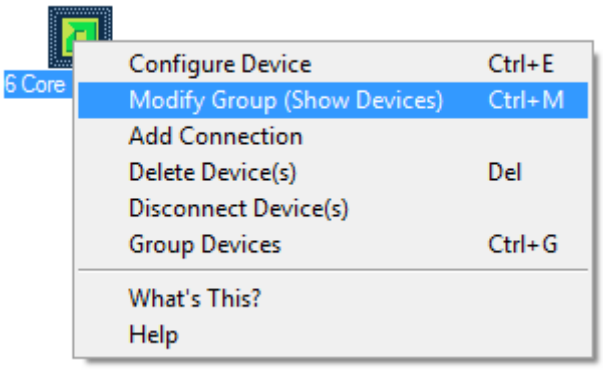


Figure 3-9: Modify Group

Click on "*Modify Group (Show Devices)*". This will open a separate show device viewer window.

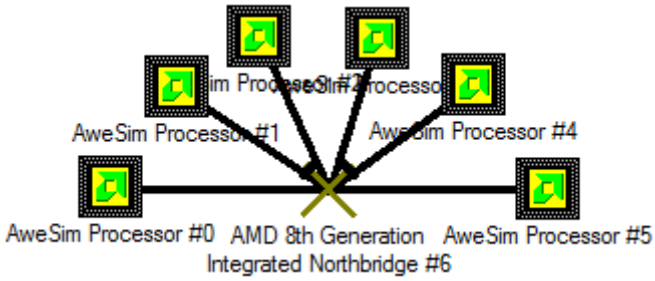


Figure 3-10: Device Group

If any modifications are done to the device group, then they will be saved with the BSD. Note that it is possible to modify a device group to a point where its children look nothing like the original device.

3.3.4 Shell Automation Commands for Device Groups

The shell automation commands that are used for a device also work for a device group. For example, shell.KnownDevices lists all known devices (both device libraries and device groups). For example, a device group exposes ports and connections, so "`shell.AvailablePorts`" and "`shell.Connect`" etc. work with a device (regardless of whether it's a group or a library).

3.3.4.1 Device Tree

You can optionally reference a device in the parent and child grouping device tree, using the syntax separator " `->` " between device parent and child, and "`-> Machine #1`" as the root device. Here are some examples, using a machine and platform that just has two "`4 core Node`" devices...

```
1 simnow> shell.createddevices
             "4 core Node #0"
             "4 core Node #1"

1 simnow> shell.CreatedDevices "-> Machine #1"
             "4 core Node #0"
               "4 core Node #1"

1 simnow> shell.createddevices "-> Machine #1 -> 4 core Node #0"
Cpu:0          "AweSim Processor #0"
Cpu:1          "AweSim Processor #1"
Cpu:2          "AweSim Processor #2"
Cpu:3          "AweSim Processor #3"
sledgenb:0     "AMD 8th Generation Integrated Northbridge #4"

1 simnow> shell.createddevices "-> Machine #1 -> 4 core Node #1"
Cpu:4          "AweSim Processor #0"
Cpu:5          "AweSim Processor #1"
Cpu:6          "AweSim Processor #2"
Cpu:7          "AweSim Processor #3"
sledgenb:1     "AMD 8th Generation Integrated Northbridge #4"
```

```
1 simnow> shell.modules
xtrsvc:0
shell:0
Cpu:0
sledgeldt:0
sledgenb:1
sledgenb:0
Cpu:1
Cpu:2
Cpu:3
sledgeldt:1
Cpu:4
Cpu:5
Cpu:6
Cpu:7
```

Notice the "`shell.modules`" list is flat, but the devices are in a tree structure that allows us to have both a "`-> Machine #1 -> 4 core Node #0 -> AweSim Processor #0`" and a "-> Machine #1 -> 4 core Node #1 -> AweSim Processor #0". Also notice that our default view ignores the tree, and just shows us two devices: "`4 core Node #0`" and "`4 core Node #1`".

3.3.4.2 Enabled vs. Disabled vs. Mixed

Shell device commands like "`shell.Location`" or "`shell.AddDevice`" have generic meanings (regardless of whether the device is a group or library). But some are defined from an aggregation of the children. For example, "`shell.GetFastPath`" can return "*Enabled*", "*Disabled*", or "*Mixed*" (means some children are "*Enabled*" and some are "*Disabled*").

```
1 simnow> shell.GetLogIO "4 core Node #0 -> AweSim Processor #0"
PCI:       Disabled
IO:        Disabled
IOfpdis:   Enabled
MEM:       Disabled
MEMfpdis:  Enabled
GETMEMPTR: Disabled

1 simnow> shell.GetLogIO "4 core Node #0 -> AweSim Processor #1"
PCI:       Disabled
IO:        Disabled
IOfpdis:   Disabled
MEM:       Disabled
MEMfpdis:  Disabled
GETMEMPTR: Disabled
```

In this example, all other child devices of "`4 core Node #0`" are "*Disabled*" for all log options.

```
1 simnow> shell.GetLogIO "4 core Node #0"
PCI:       Disabled
IO:        Disabled
```

```
IOfpdis:   Mixed
MEM:       Disabled
MEMfpdis:  Mixed
GETMEMPTR: Disabled

1 simnow> shell.GetLogIO "-> Machine #1"
PCI:       Disabled
IO:        Disabled
IOfpdis:   Mixed
MEM:       Disabled
MEMfpdis:  Mixed
GETMEMPTR: Disabled
```

3.3.5 Device Group Examples

Device groups can be a powerful building block for SimNow users. These next examples should help give further understanding about device groups, and demonstrate some practical uses.

3.3.5.1 Example: 1GB DDR2 memory

When you instantiate a "*Dimm Bank*" known device into a created device, you get its default state of 8 empty dimm's with no configuration. You can then configure the "*Dimm Bank*", such as by opening the device's GUI configuration properties to specify general options (such as max number of dimm's), and to configure each dimm (such as by importing an SPD). You could configure it, for example, to emulate a dimm bank with 2 DDR2 dimm's (1GB each).

Device groups offer us a potentially simpler alternative - for the user to instantiate a preconfigured device group. For example, we could have a device group "*Dimm DDR2 1GBx2*", which has (inside it) only one child and default archive data (state) for that child. The figure below shows that the (theoretical) known device "*Dimm DDR2 1GBx2*" has inside it a single child device "*Dimm Bank #0*" that is configured with two dimm's (type DDR2, 1GB each).

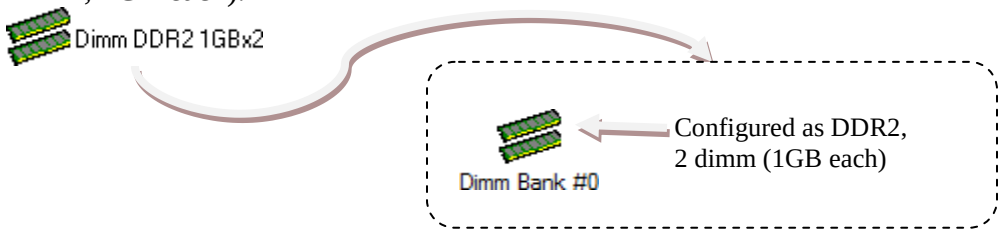


Figure 3-11: Example DIMM Device Group

When the user instantiates this (theoretical) known device "*Dimm DDR2 1GBx2*" as a created device, we get a created device "*Dimm DDR2 1GBx2 #0*" with a child device "*Dimm Bank #0*" that is already configured (as DDR2, 2 dimm, 1GB each). Our resulting main device GUI would look like this:

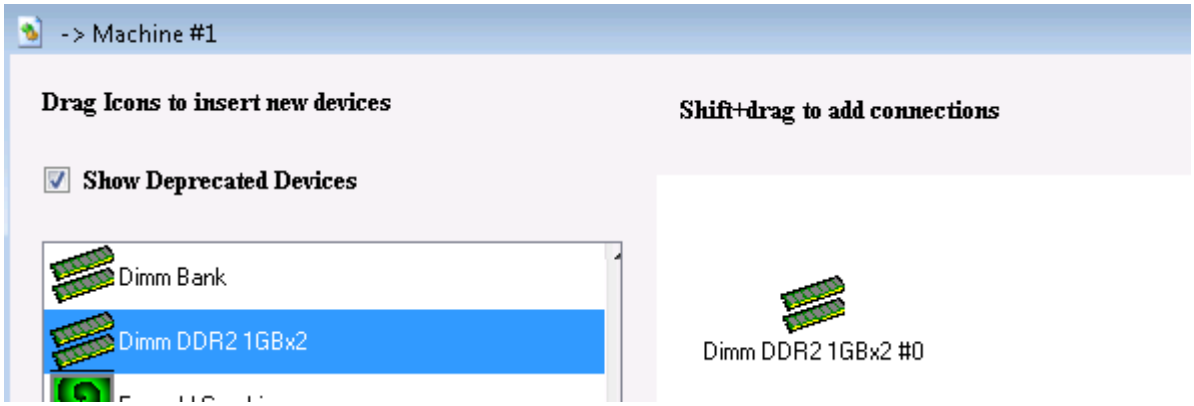


Figure 3-12: Created DIMM Device Group

The device GUI for the children of "*Dimm DDR2 1GBx2 #0*" would look like this:

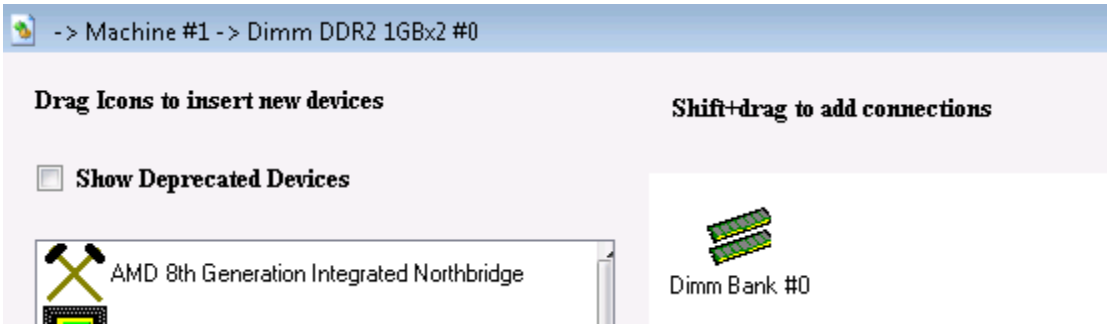

-> Machine #1 -> Dimm DDR2 1GBx2 #0
Drag Icons to insert new devices
Shift+drag to add connections
Show Deprecated Devices
AMD 8th Generation Integrated Northbridge
Dimm Bank #0

Figure 3-13: Children of DIMM Device Group

If we looked at the options and configuration of the device library "*-> Machine #1 -> Dimm DDR2 1GBx2 #0 -> Dimm Bank #0*" (either from the GUI or from the console), we would see that it is already configured as DDR2 with 2 dimm slots (1GB each).

This example demonstrates a broad concept. An existing device that has a more generic and abstract definition (such as a non-configured "*Dimm Bank*") can be wrapped in a device group to give it an identity as a particular hardware implementation (such as an already configured "*Dimm DDR2 1GBx2*"). More generally, any device can be wrapped by a device group, to give an alternate default configuration for the device's state (archive data).

3.3.5.2 Example: Quad-Core Node

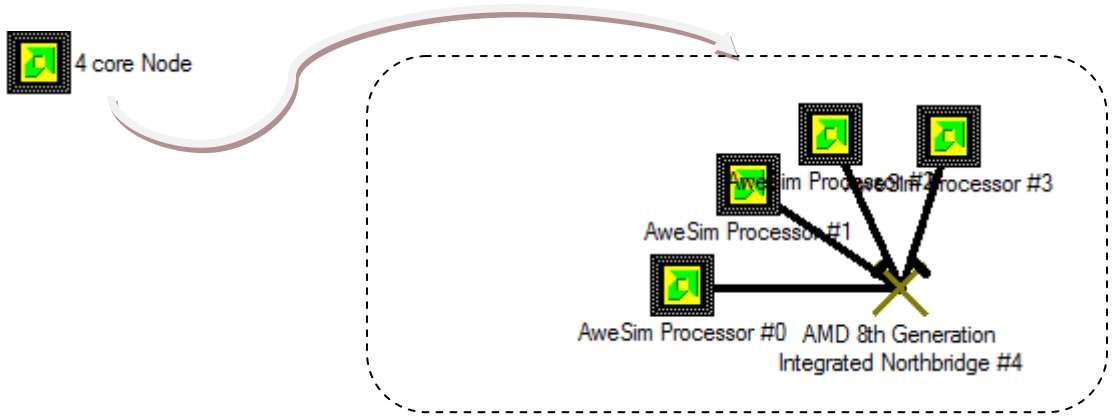
Next we will consider examples relevant to the ability of a device group to have multiple child devices, default archive data for each child device, and connections between the child devices. These next examples are based on a quad-core processor node.

Building a processor node in SimNow has traditionally been a multi-step process. First the user would add the "*AMD 8th Generation Northbridge Device*", and then add one "*AweSim Processor*" device for each processing core in the node. These devices then need to be connected together along the respective "*CPU Bus" and "Interrupt / IOAPIC*"

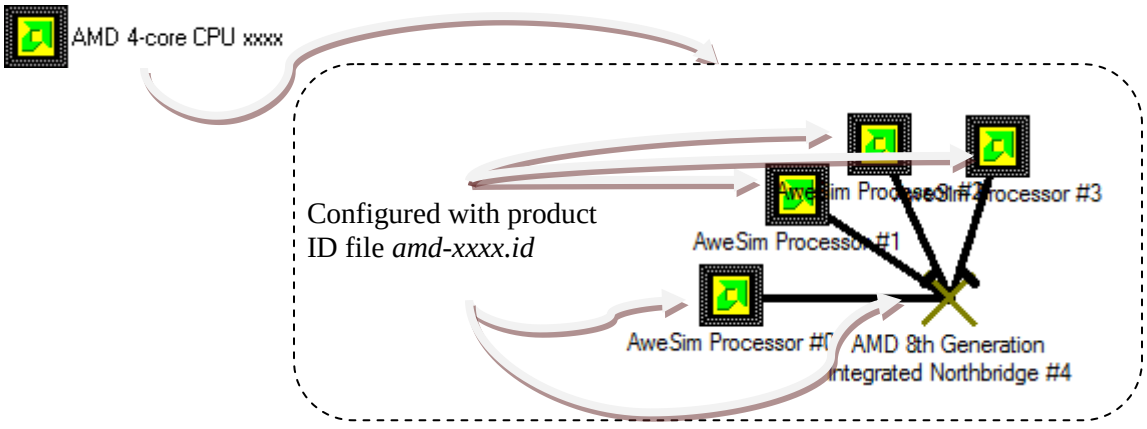
connection ports. Once the devices are connected, a user would then need to load a product ID file so that the simulated devices would represent a real and planned piece of hardware. In summary, building a Quad-core node in SimNow could take as many as 14 individual steps, and these steps would need to be repeated each time a processor node is to be added.

A device group can both simplify adding a quad-core node, and present the user with a hierarchical view. So we will give some examples with quad-core processor nodes.

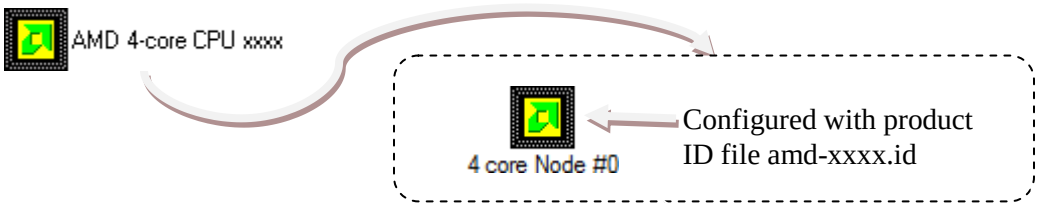
A device group is not required to specify archive data for its child devices. When such a known device group is instantiated as a created device, it simply lets its children use their own default and initial configuration state. We can create an abstract or generic "*4 core Node*" device group that does not represent a particular hardware implementation (just like a non-configured "*Dimm Bank*" does not represent a particular hardware implementation, until it is configured).



A device group can optionally specify initial and default archive data (device state) for each of its child devices. A device group with five children could specify archive data for 0, 1, 2, 3, 4, or all 5 children. We could have an "*AMD 4-core CPU xxxx*" that specifies archive data for all five of its children (configured with the (theoretical) product ID file "*amd-xxxx.id*").



This is not the only way we could create a (theoretical) "*AMD 4-core CPU xxxx*". A cleaner idea would be to reuse the non-configured abstract and generic "*4 core Node*".



This device group would (externally) be functionally the same as our previous "*AMD 4-core CPU xxxx*" example, although it has the additional layer where it cleanly reuses "*4 core Node*". We could also reuse "*4 core Node*" for other device groups that represent a particular hardware implementation of a 4-core node, such as the (theoretical) "*AMD 4-core CPU yyyy*" configured with the (theoretical) product ID file "*amd-yyyy.id*". Or a "*DeerHound RevB QuadCore Socket L1*" configured with the product ID file "*Family10hDR-L1_B0.id*".

3.3.5.3 Example: SuperIO device

For SimNow developers, device groups can be a technique for developing SimNow devices in a layered manner, promoting optimal code reuse. Before device groups were available, SuperIO devices were written as device libraries. It is cleaner to implement SuperIO device models with device groups. Typically, SuperIO devices consist of multiple functional blocks such as a UART, LPT, PS2 controller, Floppy controller etc. Device groups provide a way to develop each functional block as discrete devices that can later be *grouped* to represent a particular SuperIO controller.

3.3.6 Creating a Device Group (GUI)

From the Device Viewer window, select the devices you want to group then Ctrl + left-click a device to add or remove it from being selected; left click drag the background for a rectangle selection. The devices you select will become the children for the device group. An internal connection (see Section *3.3.1*, "*Terms*", on page *16*) will be saved inside the group. An external connection (see Section *3.3.1*, "*Terms*", on page *16*) will be maintained as a connection between the created device group and one of its sibling devices and result in an internal-to-external port mapping. Next right click one of the selected devices and choose "*Group Devices*", see *Figure 3-14*.

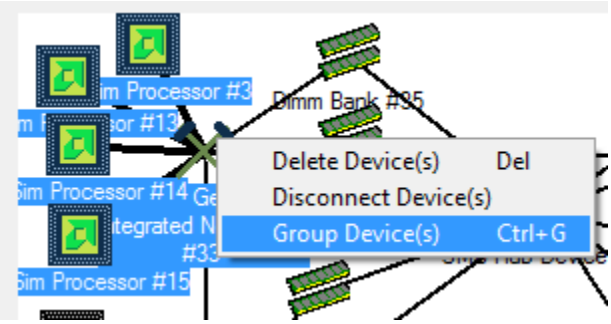


Figure 3-14: Group Devices

This brings up the device group creation wizard. On the first page, you give the group an identity as a device, by specifying device properties for the device (name, description, icon file, help file, flags). You specify a file path to save the known device group, because the wizard will create both a known device group *.bsg file, and an instance of the known device as a created device inside your current BSD (replacing the devices that you selected for grouping). The internal preview (left side) shows the child devices inside the group; the external preview (right side) shows the group as a device. This preview only shows each device icon, name, number, and internal device connections.

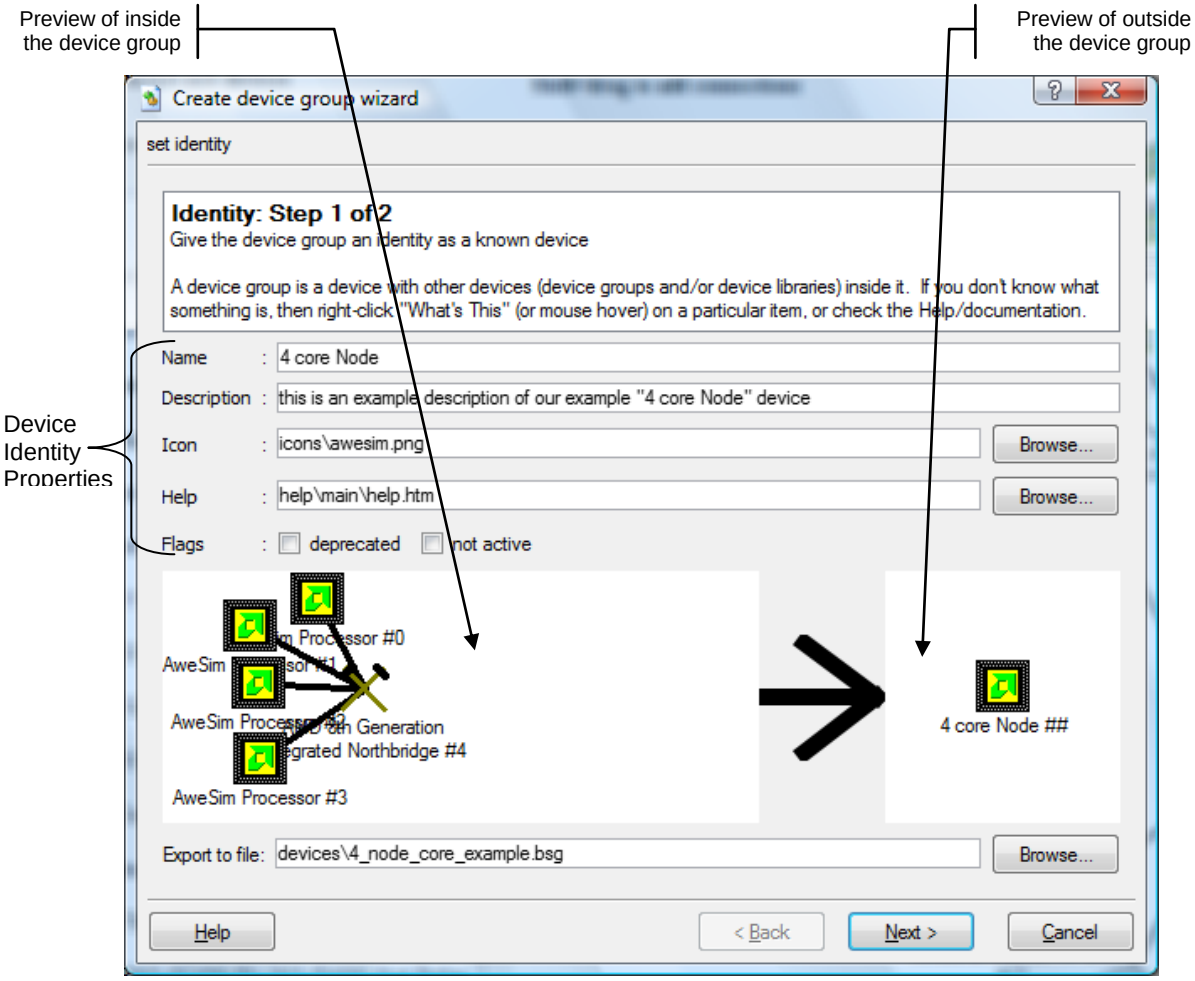

Preview of inside the device group
Preview of outside the device group
Device Identity Properties
Create device group wizard
set identity
Identity: Step 1 of 2
Give the device group an identity as a known device
A device group is a device with other devices (device groups and/or device libraries) inside it. If you don't know what something is, then right-click "What's This" (or mouse hover) on a particular item, or check the Help/documentation.
Name : 4 core Node
Description : this is an example description of our example "4 core Node" device
Icon : icons\awesim.png
Browse...
Help : help\main\help.htm
Browse...
Flags : deprecated not active
AweSim Processor #3
4 core Node ##
Export to file: devices\4_node_core_example.bsg
Browse...
Help
< Back
Next >
Cancel

Figure ?

In the second step, we specify options relative to each child device. For each child's device state, the resulting known device group can either save the child device's current state, or it can specify no default device state and thus inherit the default device state for the particular child device. For example, if a child device is an "*AweSim Processor*", we can either save the current configuration for that "*AweSim Processor*" as the default state for the known device group we are creating. Or the group's child can just inherit the defaults of the "*AweSim Processor*" known device.

For each child device, we can specify internal to external port mappings. This maps an internal port name to an external port name (a port for the device group). Since existing

external connections are maintained, we automatically require an internal to external port mapping for an existing external connection. To finish, the wizard requires that the external port names are unique to the device group, since a device must have unique port names.

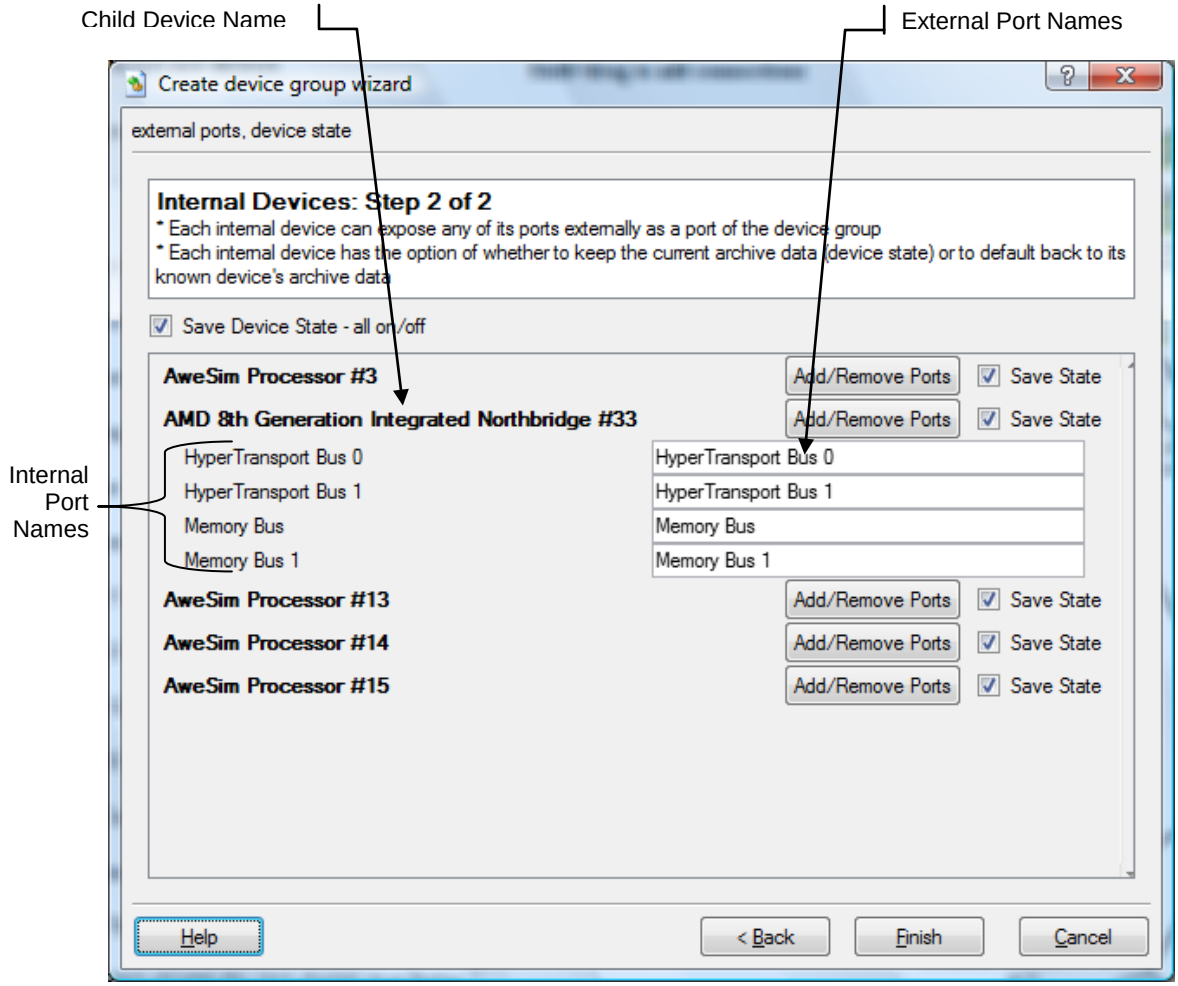


Figure ?

The "external ports, device state" page shows you all the internal to external port mappings which are currently specified for the device group. You can also click the "*Add/Remove Ports*" button for a particular child device, to open a sub-page that allows you to add and remove particular port mappings for the child device.

In a child device sub-page, each checkbox turns a particular port mapping on or off. If a checkbox is grayed out, it is because the device has an existing external connection, thus requiring the port to be mapped for the device group.

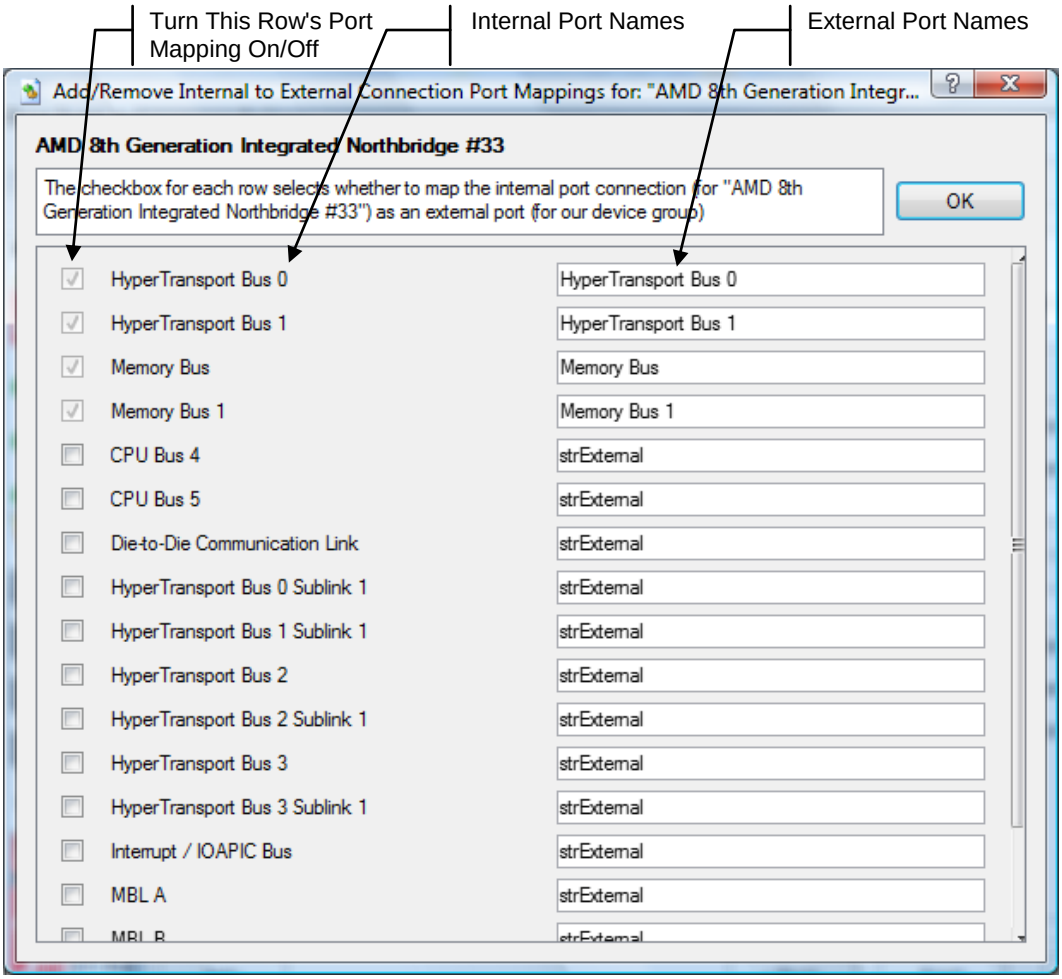

Turn This Row's Port Mapping On/Off
Internal Port Names
External Port Names
Add/Remove Internal to External Connection Port Mappings for: "AMD 8th Generation Integr...
AMD 8th Generation Integrated Northbridge #33
The checkbox for each row selects whether to map the internal port connection for "AMD 8th Generation Integrated Northbridge #33") as an external port (for our device group)
OK
HyperTransport Bus 0
HyperTransport Bus 1
Memory Bus
Memory Bus 1
CPU Bus 4
CPU Bus 5
Die-to-Die Communication Link
HyperTransport Bus 0 Sublink 1
HyperTransport Bus 1 Sublink 1
HyperTransport Bus 2
HyperTransport Bus 2 Sublink 1
HyperTransport Bus 3
HyperTransport Bus 3 Sublink 1
Interrupt / IOAPIC Bus
MBL A
strExternal

Figure ?

Clicking "*OK*" causes the "external ports, device state" page to regenerate its list of ports. So if you add a port using the checkbox, it will show up on the "external ports, device state" page.

When you are done defining the device group then simply click the "*Finish*" button. This causes the device group to get created. A known device group file is created using the *.bsg file you specified for "*Export to file*", and loaded as a known device. The devices you grouped are swapped (deleted and replaced) with a created device instance of your new device group. Its internal connections and device state come from the known device. External connections from the devices you grouped are recreated as connections to your new created device group. All of this is done automatically by the wizard when you click "*Finish*".

3.3.7 Creating a Device Group (Automation Commands)

Although it is simpler to create a device group in the GUI, it is also possible to create a device group on the console using shell automation commands. First we group a set of specified devices into an "*Unnamed Group*". Then we can customize our "*Unnamed Group*", by specifying device group options. Next we export it to a file as a known device

with a new identity as a device instead of just the generic "*Unnamed Group*". Finally, we can replace our "*Unnamed Group*" created device with a created device instance of our new known device. Here are the details of these commands:

You can specify devices to get grouped into an "*Unnamed Group*" device:

```
shell.GroupDevices[devices]
```

We can modify an existing created device group's options:

```
shell. SetDeviceGroupOption [device group] [ExternalPortMap |
ExportDeviceState] [variable args]
```

Specifically, we can add, remove, and rename the internal-to-external port mappings between a device child and its parent device group:

```
shell.SetDeviceGroupOption [device] ExternalPortMap Add [device
child] [in] [out]
shell.SetDeviceGroupOption [device] ExternalPortMap Delete [out]
shell.SetDeviceGroupOption [device] ExternalPortMap Rename [out]
[out renamed]
```

And we can specify whether or not to use the created device child's device state for each child device (for if/when the group is exported as a known device):

```
shell.SetDeviceGroupOption [device] ExportDeviceState [optional
child device] [0|1]
```

There is also a shell command to get the options (ie – to print them to the console/stdout). This can print the values for either options (ExternalPortMap or ExportDeviceState):

```
shell.GetDeviceGroupOption [device group] [ExternalPortMap |
ExportDeviceState] [variable args]
shell.GetDeviceGroupOption [device group] ExternalPortMap
[optional: child device]
shell.GetDeviceGroupOption [device group] ExportDeviceState
[optional: child device]
```

We can export a created device group (including the options we set) to a known device file. To do this, we also specify values for the known device's identity as a device:

```
shell.ExportDeviceGroupToFile [device group] [name] [desc] [icon]
[help] [flags] [bsg file path]
```

The previous command only exports the created device group to a file as a known device; it does not change our existing created device group. However, after we export our created device to a file, we can then replace our created device with an instance of the device we exported. By doing this, we give our device a new device identity:

```
shell.SwapDevice [created device] [known device]
```

3.3.8 Ungrouping a created device group

Since a device group is really just a container for its child device's, with its own identity as a device, it is simple to ungroup a device group, on either the GUI or the console. In the show devices GUI, right-click a device group, click "*Ungroup Device*". Or, in the console, execute the command:

```
shell.UngroupDevice [created device group]
```

3.4 Main Window

The AMD SimNow™ Main Window, shown in Figure 3-1, is the main application window. It contains a Menu Bar with a set of pull down menus, and a Tool Bar, both of which control many aspects of the simulation environment. The console window, shown in Figure 3-15, provides a textual interface for status information and command-line style control, see Section A.7, "*Automation Commands*", on page 224.

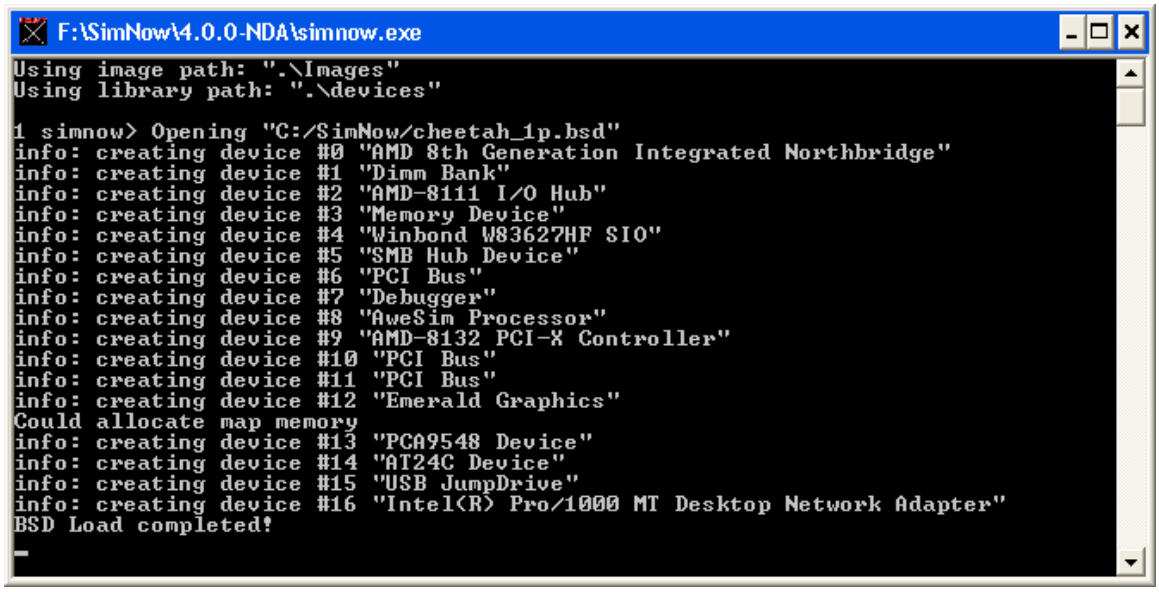

F:\SimNow\4.0.0-NDA\simnow.exe
Using image path: ".\Images"
Using library path: ".\devices"
1 simnow> Opening "C:/SimNow/cheetah_1p.bsd"
info: creating device #0 "AMD 8th Generation Integrated Northbridge"
info: creating device #1 "Dimm Bank"
info: creating device #2 "AMD-8111 I/O Hub"
info: creating device #3 "Memory Device"
info: creating device #4 "Winbond W83627HF SIO"
info: creating device #5 "SMB Hub Device"
info: creating device #6 "PCI Bus"
info: creating device #7 "Debugger"
info: creating device #8 "AweSim Processor"
info: creating device #9 "AMD-8132 PCI-X Controller"
info: creating device #10 "PCI Bus"
info: creating device #11 "PCI Bus"
info: creating device #12 "Emerald Graphics"
Could allocate map memory
info: creating device #13 "PCA9548 Device"
info: creating device #14 "AT24C Device"
info: creating device #15 "USB JumpDrive"
info: creating device #16 "Intel(R) Pro/1000 MT Desktop Network Adapter"
BSD Load completed!

Figure 3-15: Console Window

3.4.1 SimStats and Diagnostic Ports

The *SimStats* and *Diagnostic Ports* numeric displays appear in the *Main Window* when a Southbridge device is added to the workspace. The *SimStats* display shows host and simulation elapsed time and a simulation MIPS counter that is updated as the simulation runs. The simulator effectively has a built-in POST card output, ports 80h to 87h and e0h to e3h. You can see these codes on the right upper part of the *Main Window* in the "*Diagnostic Ports*" section.

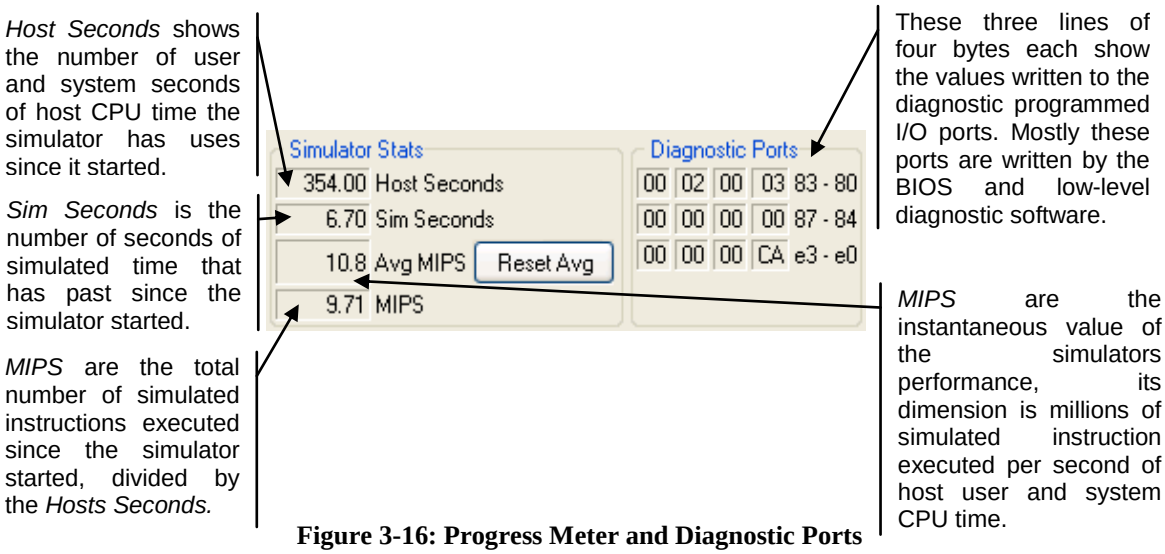

Host Seconds shows the number of user and system seconds of host CPU time the simulator has uses since it started.
Sim Seconds is the number of seconds of simulated time that has past since the simulator started.
MIPS are the total number of simulated instructions executed since the simulator started, divided by the *Hosts Seconds.*
Simulator Stats
354.00 Host Seconds
6.70 Sim Seconds
10.8 Avg MIPS
Reset Avg
9.71 MIPS
Diagnostic Ports
00 02 00 03 83 - 80
00 00 00 00 87 - 84
00 00 00 CA e3 - e0
These three lines of four bytes each show the values written to the diagnostic programmed I/O ports. Mostly these ports are written by the BIOS and low-level diagnostic software.
MIPS are the instantaneous value of the simulators performance, its dimension is millions of simulated instruction executed per second of host user and system CPU time.

Figure 3-16: Progress Meter and Diagnostic Ports

The simulation counter measures the number of microseconds of simulated time. However, it is not a performance or cycle-based simulator, so the simulated time is estimated.

3.4.2 CPU-Statistics Graphs

There are several graphs that can be displayed on the left side of the *Main Window*. These graphs can be activated by the *"View→CPU Graphs*" menu selection.

3.4.2.1 Translation Graph

The *Translation Graph* updates once a second. Full vertical scale means the address-Translation cache (tcache) is full. Dark color on the bottom of the graph represents percent of tcache containing valid translations. Lighter color above the dark color represents percent of tcache containing invalidated translations. Black color growing from the top represents the meta data that describes the translations.

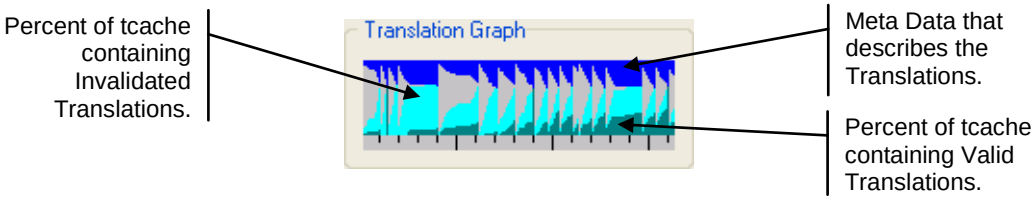

Percent of tcache containing Invalidated Translations.
Translation Graph
Meta Data that describes the Translations.
Percent of tcache containing Valid Translations.

Figure 3-17: CPU Translation Graph

3.4.2.2 Real MIPS Graph

The *Real MIPS Graph* updates once a second. If this value exceeds what can be displayed on this graph, the graph line turns red. It shows the instantaneous MIPS, i.e., how many millions of instructions per host CPU-second at which the simulator is running. A value of zero will appear as a one-pixel-high horizontal line. Full scale represents 100 MIPS.

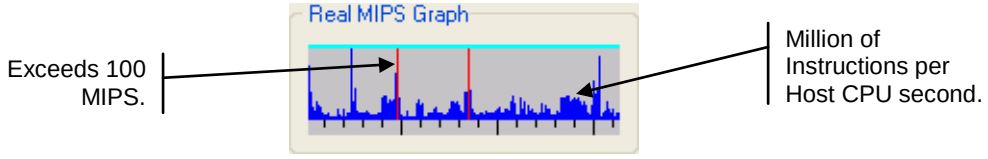


Figure 3-18: CPU Real MIPS Graph

3.4.2.3 Invalidation Rate Graph

The *Invalidation Rate Graph* updates once a second. If this value exceeds what can be displayed on this graph, the graph line turns red. A rate of zero will appear as a horizontal line, one pixel high. Full vertical scale represents one invalidatated translation per thousand simulated instructions. The lower, darker color represents plain invalidations. The upper, lighter color represents range invalidations. This upper, lighter color is a minimum of one pixel high, i.e., a value of zero range invalidations still results in a one-pixel-high line of the lighter color.

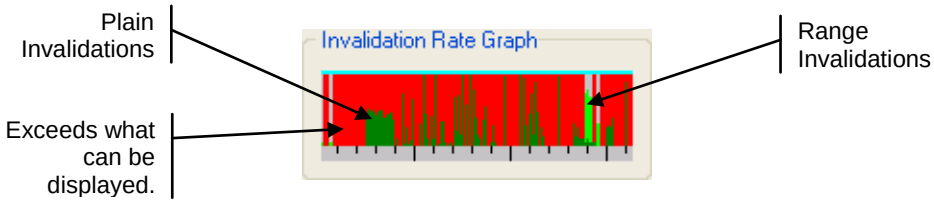


Figure 3-19: CPU Invalidation Graph

3.4.2.4 Exception Rate Graph

The *Exception Rate Graph* updates once a second. If this value exceeds what can be displayed on this graph, the graph line turns red. A rate of zero appears as a horizontal line one pixel high. Full vertical scale represents a rate of one exception taken by the simulator per ten simulated instructions. These exceptions may be internal to the simulator and not turn into exceptions in the simulated machine. The lower, darker color represents all such exceptions other than segmentation violation (SEGV) exceptions. The upper, lighter color represents all the SEGV exceptions. This upper, lighter color is a minimum of a one-pixel-high line, i.e., a value of zero SEGV exceptions still shows a one-pixel-high line of the lighter color.

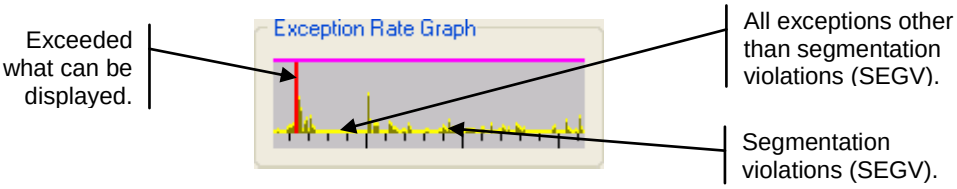

Exceeded what can be displayed.
Exception Rate Graph
All exceptions other than segmentation violations (SEGV).
Segmentation violations (SEGV).

Figure 3-20: CPU Exception Rate Graph

3.4.2.5 PIO Rate Graph

The *PIO Rate Graph* updates once a second. If the port I/O (PIO) rate exceeds what can be displayed on this graph, the graph line turns red. A rate of zero will appear as a horizontal line one pixel high. Full scale represents one PIO per ten simulated

instructions. Darker color on the bottom of the graph represents the read PIO's, the lighter color represents the write PIO's.

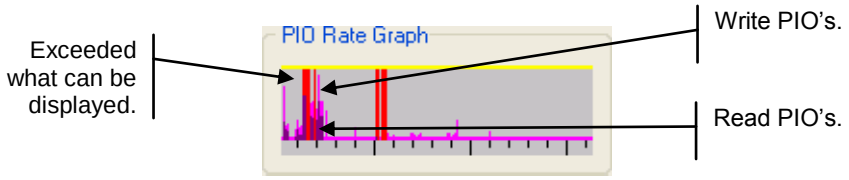


Figure 3-21: CPU PIO Rate Graph

3.4.2.6 MMIO Rate Graph

The *MMIO Rate Graph* updates once a second. If the memory-mapped I/O (MMIO) rate exceeds what can be displayed on this graph, the graph line turns red. A rate of zero will appear as a horizontal line one pixel high. Full vertical scale represents one MMIO per ten simulated instructions. Darker color on the bottom of the graph represents the read MMIO's, the lighter color represents the write MMIO's.

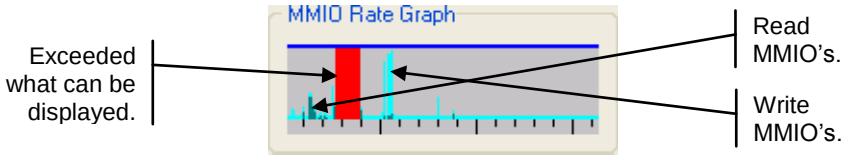


Figure 3-22: CPU MMIO Rate Graph

3.4.3 Simulated Video

The simulated video area of the *Main Window* depicts the VGA output screen that appears when a VGA device is added to the workspace. When the mouse focus is over the video area, the simulator captures host keyboard input, enabling you to type most keyboard entries on your real keyboard. This is a convenience and may not accurately position the mouse or grab all keys correctly. For more accurate mouse and keyboard capture, see "*Grab the mouse and keyboard*" in Section 5.2.3, "*Interaction with the Simulated Machine*", on page 45.

You can also allow the simulator to take complete control of the mouse and keyboard by selecting "*Special Keyboard→Grab Mouse and keyboard*". To return from this mode, press and hold *Ctrl* then *Alt*, and then release them in reverse order.

3.4.4 Hard Disk and Floppy Display

The *IDE Primary byte counts*, *IDE Secondary byte counts*, and *Floppy disk byte counts* displays appear when a Southbridge device is added to the workspace.

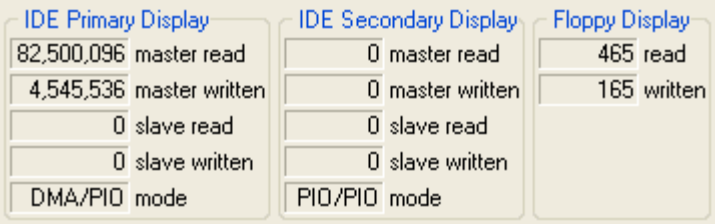


Figure 3-23: Primary, Secondary, and Floppy Displays

When a disk is accessed in simulation, the status information is updated.

3.4.5 Using Hard Drive, DVD-/CD-ROM and Floppy Images

Section 4 on page 35 describes how to create disk images. To use a disk image created by DiskTool go to the *Main Window File Menu* and choose one of the "*Set [...] Image*" menu items. This brings up an open-file dialog. Select your drive image and click on 'Ok'. Standard file extensions for disk images are shown in Table 3-3.

Image Type	File Extension
Hard Drive Image	*.hdd
Floppy Drive Image	*.fdd
DVD-/CD-ROM Image	*.iso
Generic Image	*.img

Table 3-3: Image Types

After an image is selected, any changes to the image are stored in journal form in the "*.BSD*" file, unless journaling is disabled in the Southbridge (for hard drive images) or SuperIO (for floppy drive images) device. If journaling is disabled, changes are stored to the image file, see also Section 5.2.1, "Assigning Disk-Image", on page 42.

3.4.6 Help, Problems and Bug Reports

The simulator has HTML on-line help and documentation, with "*Help*" menu entries or buttons on the dialogs. In the device view, every device has a context menu (right-click) with "*Help*" documentation links and "*What's this*" floater text.

In addition to any other support channel you may have, we encourage feedback on any problems encountered. Please send an email to simnow.support@amd.com.

4 Disk Images

The simulator uses hard-drive images to provide simulated hard disks to the simulated computer. There are several ways to obtain hard drive-images.

- Install your OS onto a hard drive in a real system, then move it to the secondary drive in a system and use *DiskTool* to copy the contents of the drive to an "*.hdd*" image file.
- Make a blank hard-drive image and a DVD-/CD-ROM "*ISO*" image, and install a fresh operating system onto the hard-drive image. To make the hard drive and DVD-/CD-ROM images, refer to Section 4.1, "*Creating A Blank Hard-Drive Image*" and Section 13, "*DiskTool*", on page 161.
- To use a physical DVD-/CD-ROM:
  - Click on the button or select "*View→Show Devices*" to open the *Device Window* (Figure 3-2, on page 9).
  - Open the Southbridge's properties window by double-clicking on it, and choose the "*HDD Secondary Channel*" tab.
  - On a Windows host type *"\\.\D:*" where "*D:*" is the drive letter for the DVD-/CD-ROM, and on a Linux host type "/dev/cdrom" in the *"Master Drive - Image Filename*" field.
  - Check the *DVD-ROM* check box below the *Filename* field.

The simulator can access media via the following mechanisms:

- IDE Hard Disk:
  - DiskTool IDE hard-disk image, is a flat file consisting of a 512-byte header (the IDE probe sector) and a raw image of data from the hard disk (if the raw data is cut off before the end of the disk, the disk-image from there on will just read as zero).
- IDE DVD-ROM: (The simulator does not simulate DVD-ROM "*insert*" events)
  - DVD-ROM disk image is a flat file of the raw image of a data DVD-/CD-ROM. These correspond exactly to ISO file images, for example.
  - IDE DVD-ROM direct access
- Floppy Disk:
  - Floppy-disk image, a flat file of the raw image of a floppy disk.
  - Floppy direct access

Please refer to Section 13, "*DiskTool*", on page 161 to find out how to set up a Windows or Linux hard-drive image for the simulator.

4.1 Creating A Blank Hard-Drive Image

To create a hard-drive image use DiskTool. You can start DiskTool by launching "*disktool.exe*" in your install directory. For convenience, you can create a desktop shortcut to launch DiskTool. When you run DiskTool, you will see the DiskTool dialog

window, as shown in Figure 4-1. It will also open a shell window, as shown in Figure 4-2, that is used to inform the user about all physical drives which DiskTool has detected.

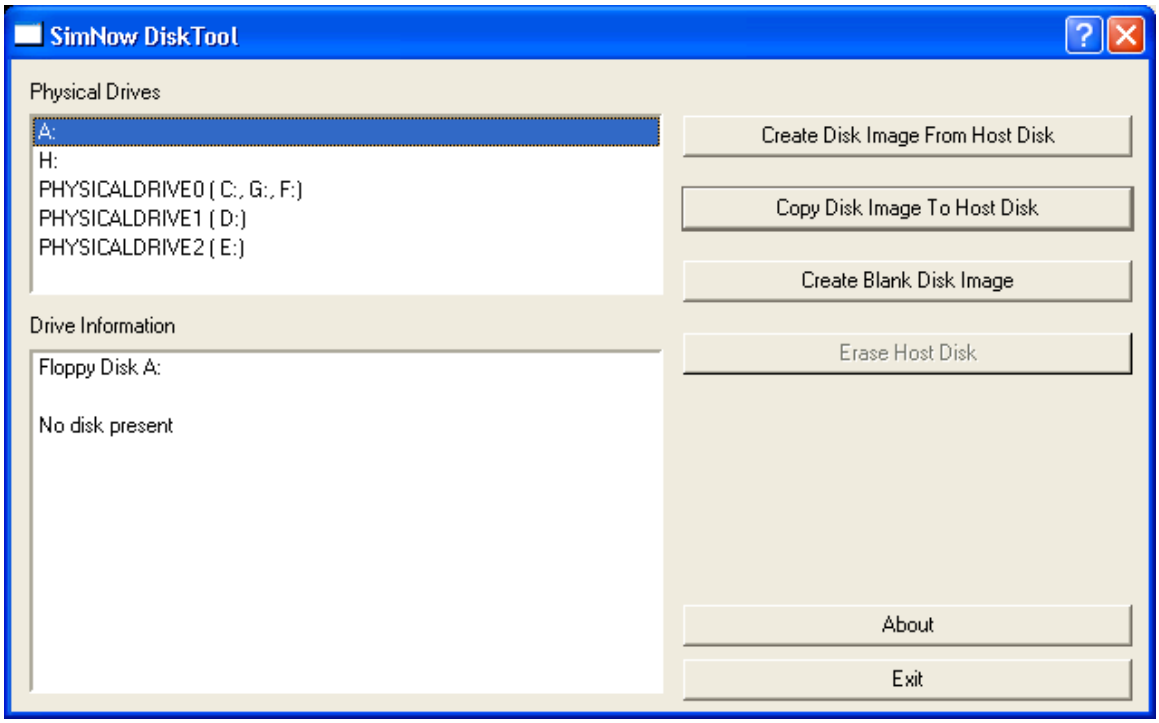


Figure 4-1: DiskTool Dialogue Window

For information about supported options and modes that DiskTool supports, please refer to Section 13, "*DiskTool*", on page 161.

Figure 4-2 shows the DiskTool shell window.

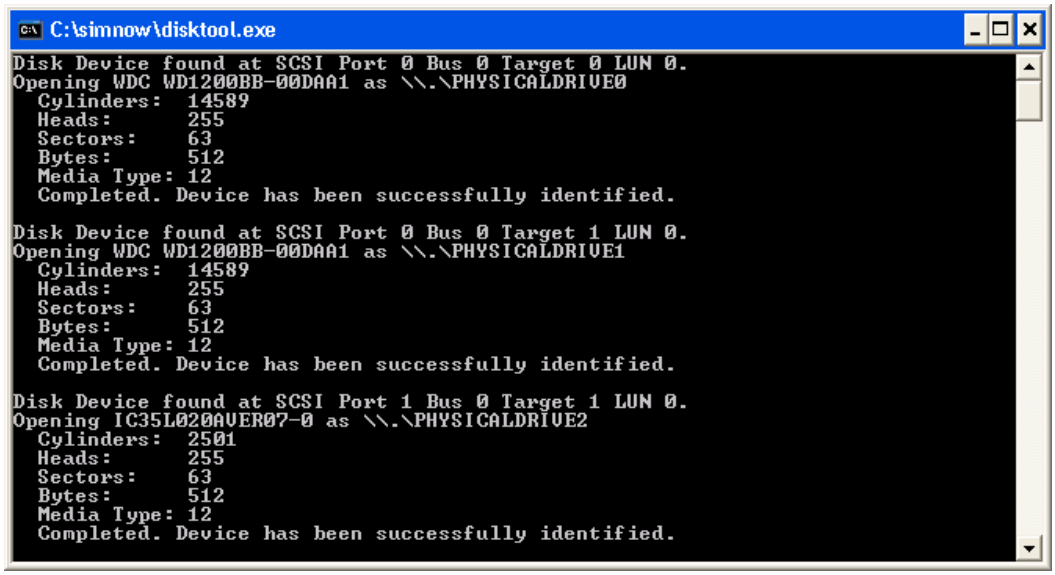

C:\simnow\disktool.exe
Disk Device found at SCSI Port 0 Bus 0 Target 0 LUN 0.
Opening WDC WD1200BB-00DAA1 as \\.\PHYSICALDRIVE0
Cylinders: 14589
Heads: 255
Sectors: 63
Bytes: 512
Media Type: 12
Completed. Device has been successfully identified.
Disk Device found at SCSI Port 0 Bus 0 Target 1 LUN 0.
Opening WDC WD1200BB-00DAA1 as \\.\PHYSICALDRIVE1
Cylinders: 14589
Heads: 255
Sectors: 63
Bytes: 512
Media Type: 12
Completed. Device has been successfully identified.
Disk Device found at SCSI Port 1 Bus 0 Target 1 LUN 0.
Opening IC35L020AVER07-0 as \\.\PHYSICALDRIVE2
Cylinders: 2501
Heads: 255
Sectors: 63
Bytes: 512
Media Type: 12
Completed. Device has been successfully identified.

Figure 4-2: DiskTool Shell Window

To create a blank disk image click on the "*Create Blank Disk Image*" button on the right side of the DiskTool dialog window (see Figure 4-1). A "*Save As*" dialog will ask you for the location and image filename that will be created. Choose the location where you want to store the blank image file and then enter the image filename. Click on the "*Save*" button. An additional dialog, see Figure 4-3, is presented that allows you to select how large the blank image file should be.

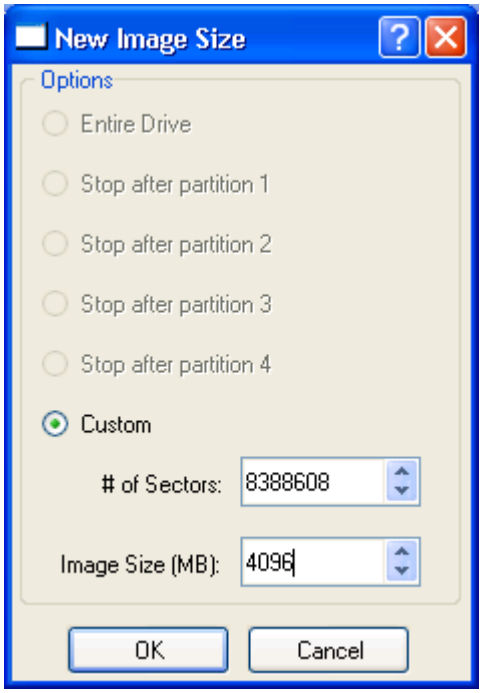


Figure 4-3: New Image Size

Before you start creating a new blank disk image make sure that the image will be large enough to install Windows or Linux on it. You can enter the image size in MB or in number of sectors. We recommend an image size of 4-GB. Increase the value of "*Image Size (MB)*" to 4096 and then click on the "*Ok*" button to create the image file. A progress bar will inform you of the progress being made (see Figure 4-4).

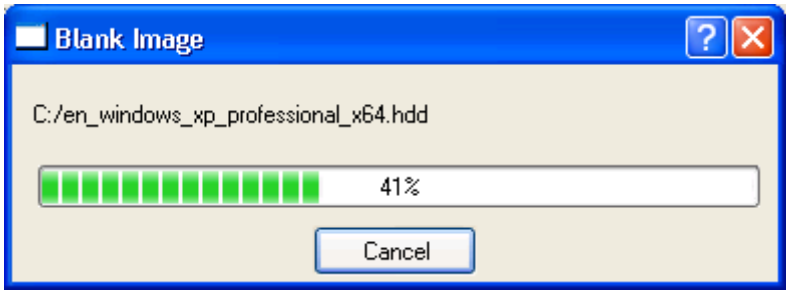


Figure 4-4: Create Blank Image

Once the image is created successfully DiskTool will display a message box, as shown in Figure 4-5. Click on the "*Ok*" button.

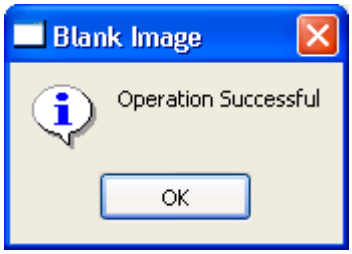


Figure 4-5: DiskTool Operation Successful

To exit DiskTool click on the "*Exit*" button on the right side of the DiskTool dialog window (see Figure 4-1).

5 Running the Simulator

You can start AMD SimNow™ by launching "*SimNow.exe*" in your install directory. For convenience, you can create a desktop shortcut to launch the simulator. When you run the simulator, you will see the simulator's *Main Window* as shown in Figure 5-1. It will also open a console window (shown in Figure 3-15) that is used for text interaction.

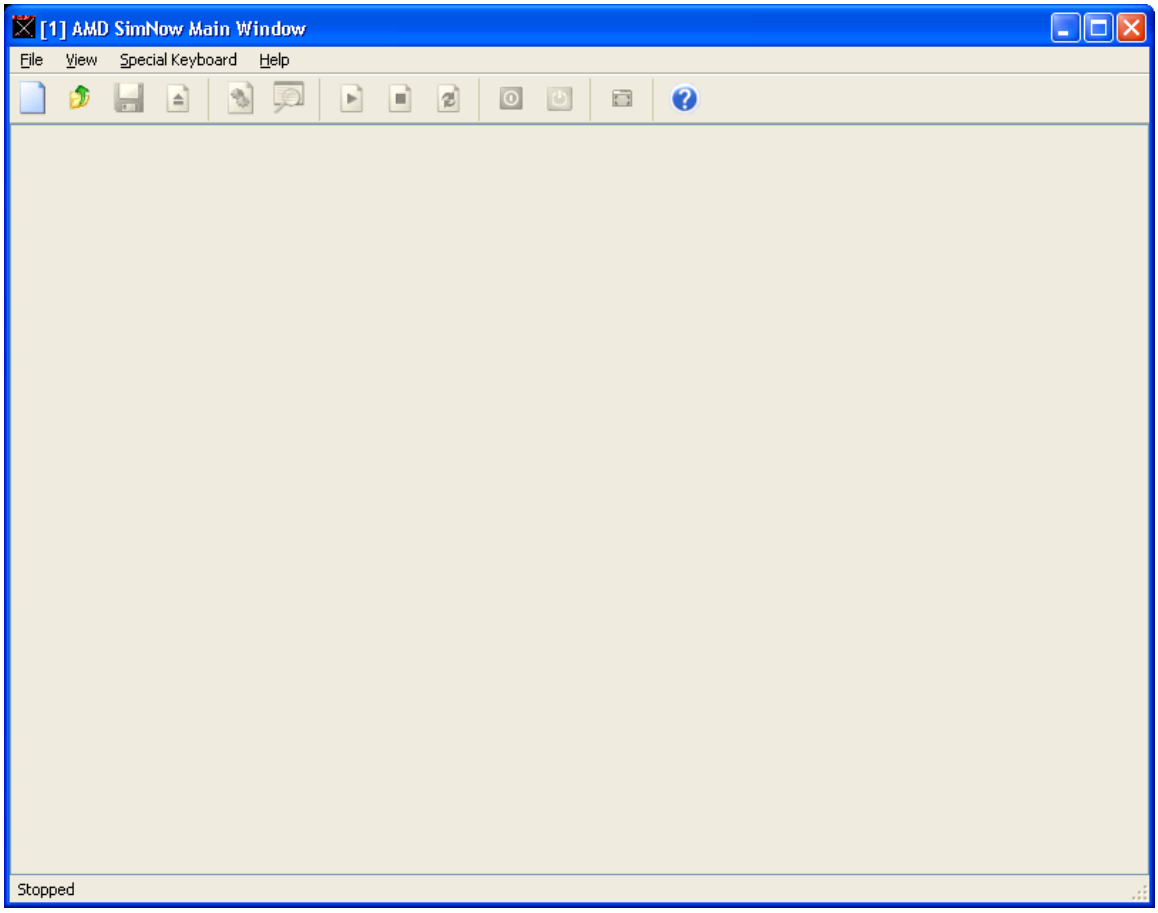


Figure 5-1: Main Window (No BSD Loaded)

5.1 Command-Line Arguments

This section describes the command-line arguments supported by the simulator. Table 5-1 shows the command-line arguments.

Argument	Description
-l <path>	Directory to load devices from. If used, it must be first.
-f <file>	Open the .bsd file <file>.
-e <file>	Execute commands in <file> on startup.
-i <path>	Image search path for loading image files.
-m <path>	Mediator connection string for network adapters to use.

Argument	Description
-n --novga	Disable VGA Window.
-c --nogui	Disable GUI (console mode).
-d	Disable mouse and keyboard inputs to simulator.
-r --register	Register the simulator with the O/S as an automation server.
-h --help -?	Print this help message.

Table 5-1: Command-Line Arguments

For instance, to open the *cheetah_1p.bsd* when starting the simulator you can enter the following:

```
C:\SimNow\simnow –f cheetah_1p.bsd
```

5.1.1 Open a Simulation Definition File

Click on and select one of the *".bsd*" files located in the "*\SimNow*" directory. The *".bsd*" files contain pre-configured simulation definitions designed to model a specific AMD processor-based computer system. For this example, load the "*cheetah_1p.bsd*" file, from in the SimNow directory. Upon loading the BSD file, the *Main Window* (shown in Figure 5-2) will be filled with three sections. The left column contains informational graphs if selected (see Section 3.4.2, "*CPU-Statistics Graphs*", on page *30*), the top row contains numeric displays of simulation statistics and disk-drive access information, and the remainder contains the *Simulation Display Area* of the simulated machine. The *Simulation Display Area* remains blank until the simulated machine is started.

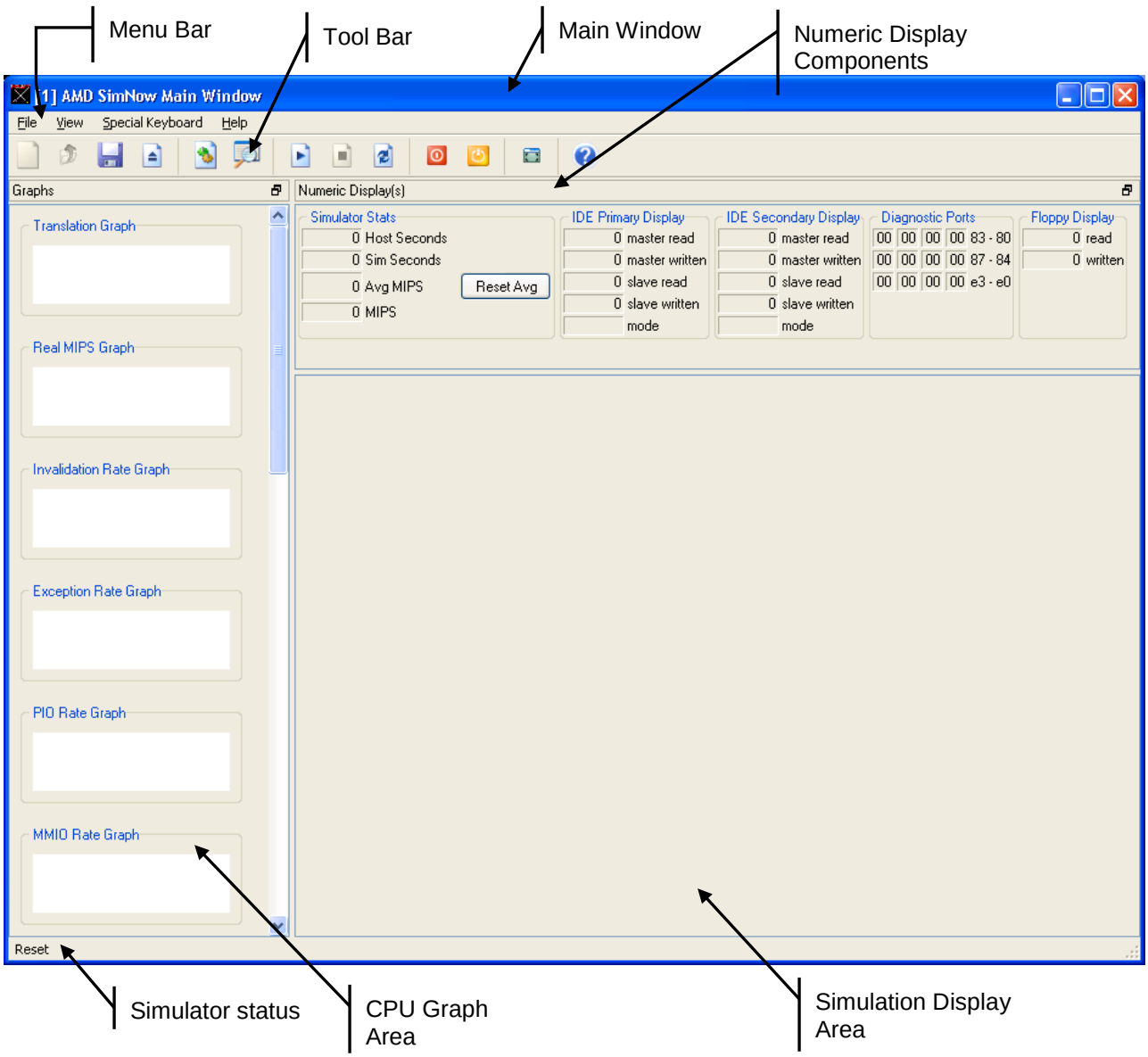


Figure 5-2: Main Window (BSD Loaded)

You can view the configuration of the simulated machine by clicking on . A window appears with a graphical representation of the simulated machine, as shown in Figure 5-3.

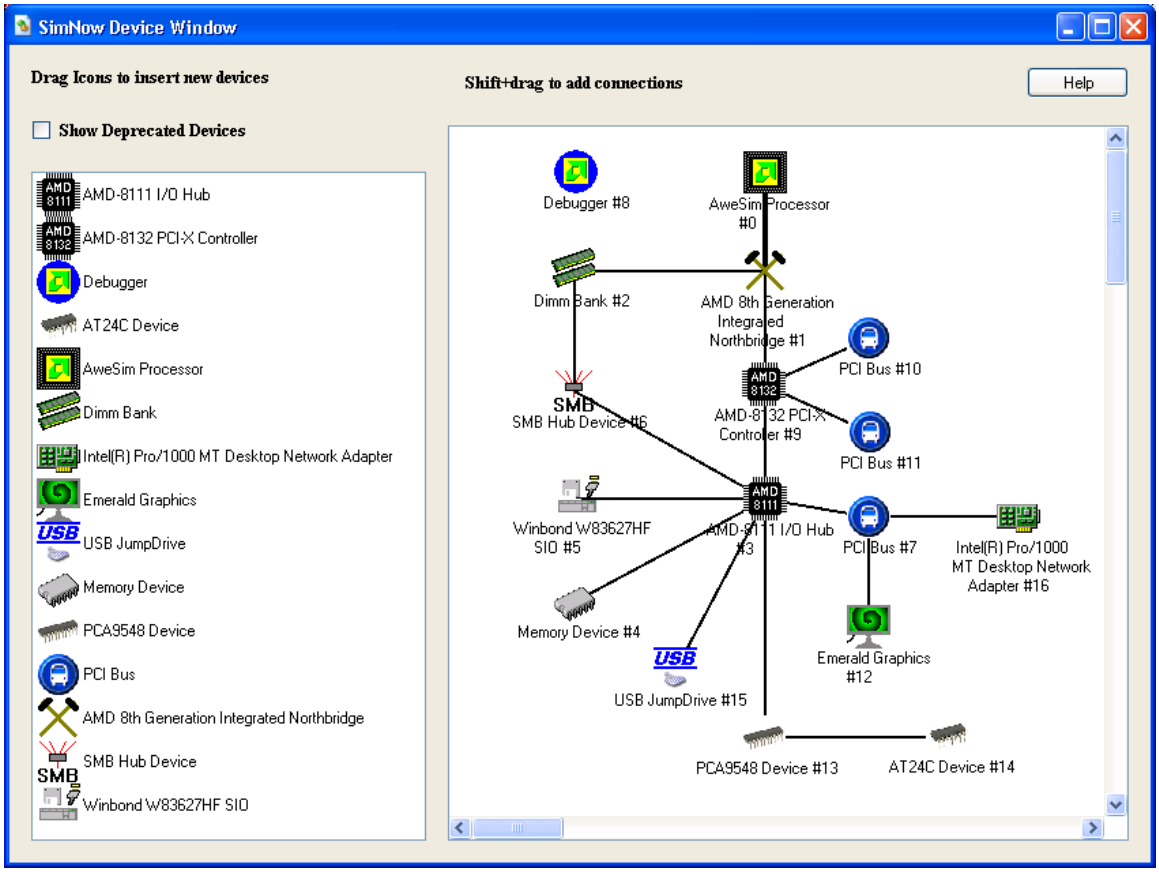


Figure 5-3: Device Window

5.2 Installing an Operating System

This section describes the steps that are necessary to install Windows or Linux using the simulator. Before you can start installing an operating system make sure you have a blank hard-drive image available. To create a blank hard-drive image with DiskTool please follow the steps in Section 4.1, "*Creating A Blank Hard-Drive Image*", on page 35.

5.2.1 Assigning Disk-Images

Assign a blank hard-drive image by selecting "*File→Set IDE Primary Master Image...*". Open the directory that contains your hard-drive images and choose a blank hard-drive image that you created earlier (see Section 4.1, "*Creating A Blank Hard-Drive Image*", on page 35) or use one of the hard-disk images which come with the simulator (see Section A.2.4.1, "*Hard-Disk Image Files*", on page 179) and un-check the "*Journal*" check-box (see below "*The IDE controller has two important features*"), then click on "*Ok*".

Assign the first OS installation ISO image to the IDE Secondary Master Channel of the hard-disk controller by selecting "*File→Set IDE Secondary Master Image...*".

If you don't have access to any ISO images you have two options:

- You can download Linux ISO images from fedora.redhat.com. If you are a MSDN Subscription member you can also download Windows ISO images from Microsoft's MSDN Subscription Webpage.
- You can assign a physical host DVD-/CD-ROM drive to the simulators IDE Secondary Master Channel and use your host's physical DVD-/CD-ROM drive to install from a CD or DVD media. Section 4, "*Disk Images*", on page 35 describes how to assign a physical DVD-/CD-ROM drive

When the OS installation prompts you, eject the current ISO image using "*File→Clear IDE Secondary Master*" and insert the next ISO image using "*File→Set IDE Secondary Master*". In case you are using a physical DVD/-CD-ROM drive for the OS installation, eject the media and insert the next media.

The disk-images are now assigned to the device that is connected to the IDE Primary Master and IDE Secondary Master connector of the hard disk controller, as shown in Figure 7-22 on page 93.

The IDE controller has two important features:

- All disk devices (Primary Master, etc.) by default have the disk journaling feature turned on, which allows simulations to write to the disk image during normal operation and not affect the contents of the real disk image. This is useful for being able to kill a simulation in the middle, for multiple copies of the simulator running at the same time, etc. Journal contents are saved in BSD checkpoint files but lost if you don't save a checkpoint before exiting. To change journal settings or commit journal contents to the hard disk image, go to the *Device View Window*, then the AMD-8111™ Southbridge, then the configuration for the hard disk in question on either the Primary or Secondary IDE controller. Here you can either commit the contents of the journal to the hard-disk image or turn off journaling for the hard disk image in question. Turning off journaling is recommended during the installation process for an operating system.

- DVD-ROM support is provided through an option in the BSD platform checkpoint file. To install a DVD-ROM at any hard disk device location (Secondary Master, Primary Slave, etc.), turn on the '*DVD-ROM*' checkbox. By default, the Secondary Master in all distributed BSDs has '*DVD-ROM'* checked and is a DVD-ROM device.

Copying files into the simulator corresponds to putting data into some media on the Host which will be inserted into the simulation. The choices for doing this are:

- Create an ISO image with the data inside it then get it into your guest OS. Use the "*File→Set IDE Secondary Master Image*" item in the *Main Window Menu* to insert it into the DVD-ROM simulation, which is by default on the secondary master position in all BSDs. Finally, mount it in your guest OS.

- Use a raw floppy-disk image in a manner similar to the above. It's a lot smaller and a bit more hassle, so we don't recommend it.
- Mount a hard-disk image on the host. (On a Linux host, you can use the "loopback device").
- Use the JumpDrive USB device to copy files into the simulator and out of the simulator, see. Section A.7.26, "*JumpDrive*", on page 244.

Copying files out of the simulator corresponds to putting some data into some media in the guest which will then be extracted on the host. To do this, mount a hard-drive image on the host after placing the data on it in the guest. (On a Linux host, you can use the "*loopback device*").

5.2.2 Run the Simulation

Once the disk-images are assigned, the simulation may be started by clicking on the *Play* button on the *Main Window's Tool Bar*.

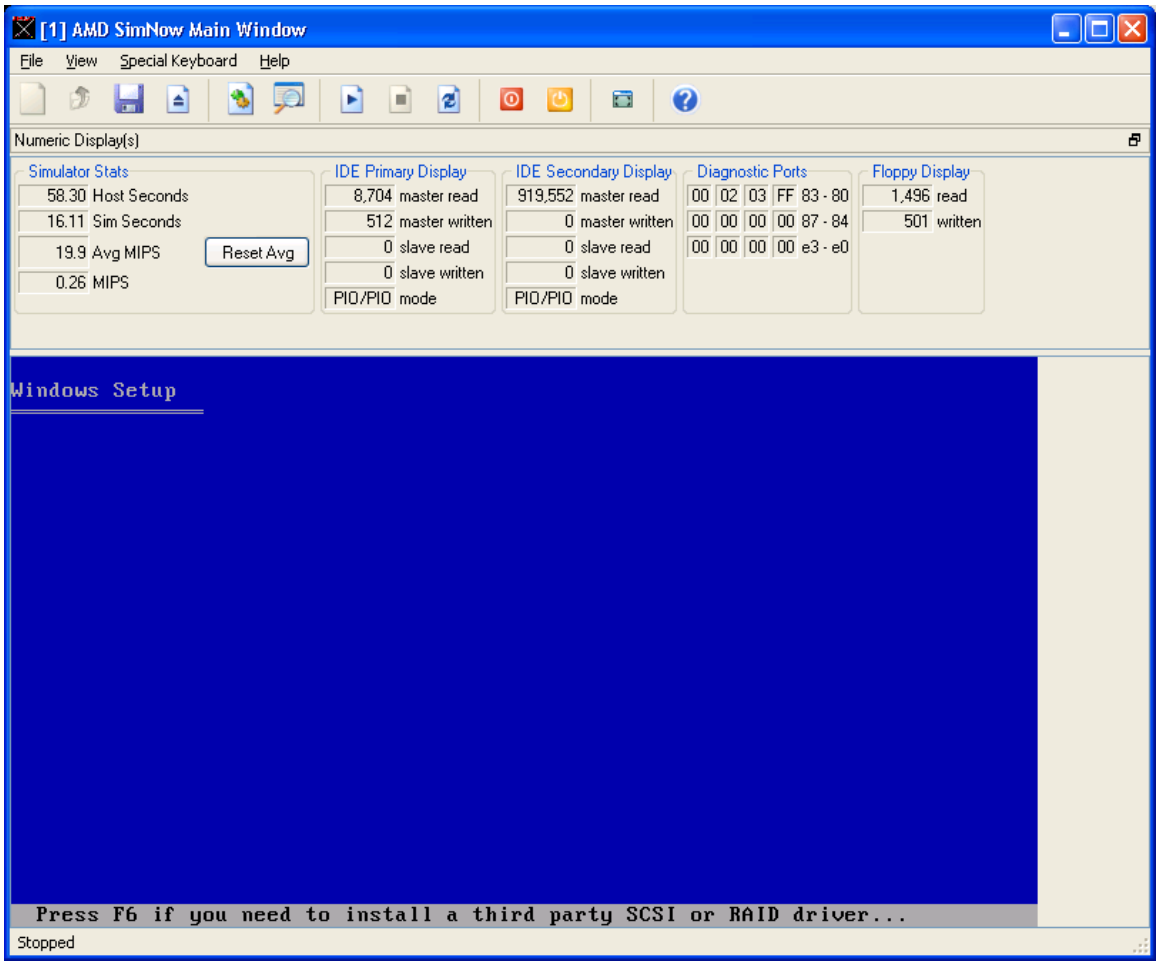


Figure 5-4: Installing WindowsXP

5.2.3 Interaction with the Simulated Machine

The simulator will boot and the simulated output screen appears in the bottom right portion of the *Main Window*, which is the *Simulation Display Area*. When the focus is on this portion of the window, most keystrokes and mouse operations are passed through to the simulated machine. When moving the mouse cursor outside of the Simulation Display area the *Main Window* returns the mouse cursor and keyboard control to the host machine. Some keystrokes, such as ALT-combinations, must be entered using the Special Keyboard Menu. At present we have some predefined keystrokes which can be entered by the *"Special Keyboard→Special Keys Generator"* as shown in *Figure 5-5*. The simulator superimposes a small square over the screen at the position of the host mouse. You can also allow the simulator to take complete control of the mouse and keyboard by selecting "*Special Keyboard→Grab Mouse and Keyboard*". To return from this mode, press and hold *Ctrl* then *Alt*, and then release them in reverse order.

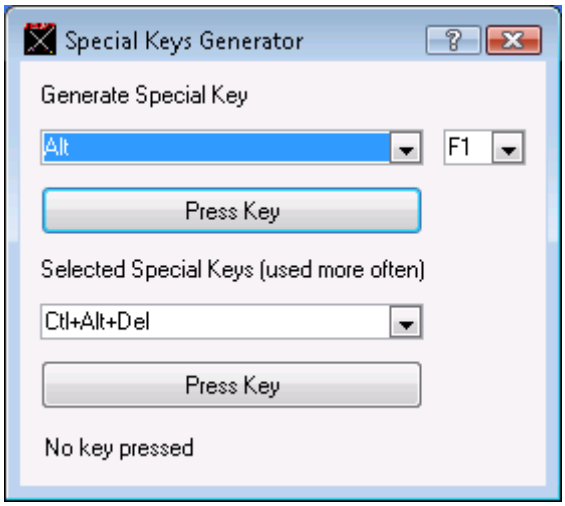


Figure 5-5: Special Keys Generator

5.2.4 Simulation Reset

To reset the entire simulator, stop the simulation with the "*Stop*" button ( ), then press the "*Reset*" button ( ), which is to the right of the "*Stop*" button. At this point, hard-drive images may be changed as described in 5.2.1 Assigning Disk-Image on page 42.

5.3 Multi-Machine Support

The multiple machine concept allows the simulator to create multiple simulation machines within the same process space, and to load and execute these machines independently.

The default shell provided with the simulator includes three new commands that allow the user access to the multiple machine functionality.

The '*newmachine*' command creates a new 'emtpy' simulation machine. The created new machine is in no way related to the current machine. Tou can load BSDs, edit device

configurations, etc., in the new machine, and they are completely independent of any other 'machine' currently loaded.

The leading number before the prompt identifies which machine is currently the active machine. All subsequent automation commands typed into the console window are directed to the current machine.

Table 5-2 describes the arguments provided by the *newmachine* command.

Argument	Description
--nogui	Disable Graphical User Interface (GUI).
--gui	Enable Graphical User Interface (GUI).
-c	Enable console mode.
--novga	Disable VGA Window.
--vga	Enable VGA Window.
-n	Disable VGA Window.
-d	Disable mouse and keyboard inputs to simulator.
+d	Enable mouse and keyboard inputs to simulator.
-i <path>	Image search path for loading image files.
-m <path>	Mediator connection string for network adapters to use.
-l <path>	Directory to load devices from. If used, it must be first.

Table 5-2: Newmachine Command Arguments

```
Usage:

newmachine[ [--nogui | -c | --gui] [--novga | -n | --vga]
            [-d | +d] [-i <path>] [-m <path>] [-l <path>] ]
```

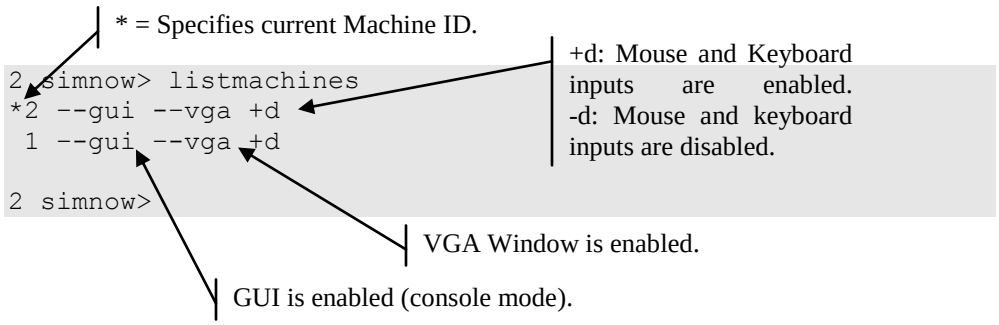
The following command creates a new simulation machine:

```
1 simnow> newmachine

2 simnow>
```

The '*switchmachine n*' command switches the console window to the machine identified by '*n*'. All subsequent automation commands typed into the console window are directed to the given machine '*n*'.

```
2 simnow> switchmachine 1

1 simnow>
```

The '*listmachines*' command lists all machines that currently exist.


* = Specifies current Machine ID.
2 simnow> listmachines
*2 --gui --vga +d
1 --gui --vga +d
2 simnow>
+d: Mouse and Keyboard inputs are enabled.
-d: Mouse and keyboard inputs are disabled.
VGA Window is enabled.
GUI is enabled (console mode).

See also Section 5.1, "*Command-Line Arguments*", on page 39 for further information regarding available command-line arguments.

.

To exit a created simulated machine enter '*exit*', as shown in the following example:

```
1 simnow> exit

2 simnow>
```

This example exits the simulated machine '*1*'.

This page is intentionally blank.

6 Create a Simulated Computer

This section describes how to create a simulated computer from scratch. We will build a computer identical to the "s*olo.bsd*" computer.

Figure 6-1 shows the layout of the existing "s*olo.bsd*" Device Window. The device position is not important because the connections between devices are completely represented by the lines between devices.

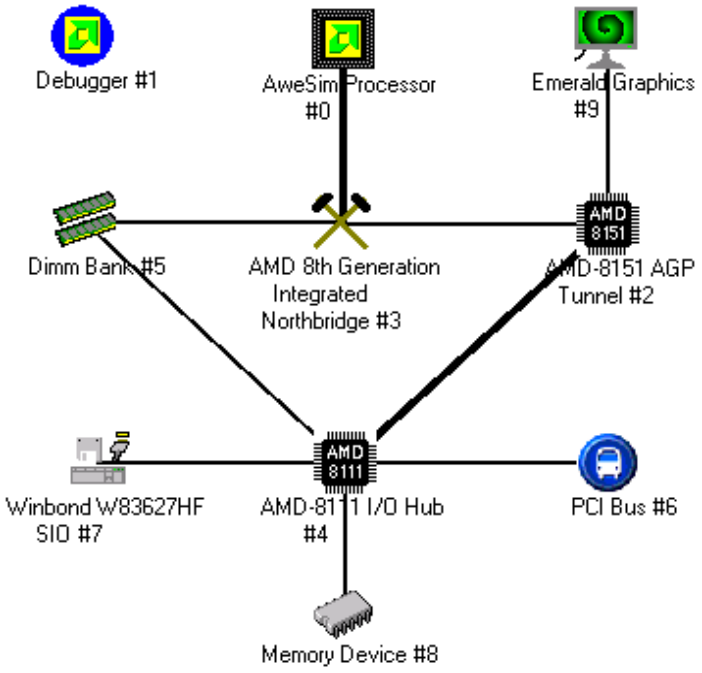


Figure 6-1: Solo.bsd Configuration

The thickness of the connection between devices represents the number of existing connections.

6.1 BSD Files

A BSD file contains the configuration of a computer system (how models are connected together and their settings), sometimes called a "virtual motherboard description" and a checkpoint of the state of all devices in the simulator. BSD files are stored in the simulator's home directory. For a list of BSD files provided with the simulator, see Appendix A.2.1 on page 178.

6.2 Device Placement

To place a device into a simulated computer system:

1. Open a new simulator instance by launching "*SimNow.exe*" in your install directory.
2. Select "*File→New BSD*" or click on the button to create a new BSD file.

3. Select "*View→Show Devices*" or click on the button to show the blank *Device Window*.
4. For each item added, click and drag the icon from the device list on the left side into the workspace area on the right side of the window.
5. Add the Debugger device. This device needs no connections drawn.
6. Add the AweSim Processor and the AMD 8th Generation Integrated Northbridge. When you add the AweSim Processor, CPU Simulation Stats are added to the *Main Window*.
7. Connect the AweSim Processor and the AMD 8th Generation Integrated Northbridge by shift-click-dragging from one to the other. When the "*Connections*" tab of *Device Properties Window* appears (shown in Figure 6-2), choose the CPU Bus 0 for both devices, and click on Ok. The connection appears as a line between the two devices on the *Device Window*. Then create an additional connection between the two devices using the Interrupt/IOAPIC Bus on each device. The *Device Window* shows only one line for the two connections between these devices. You can view the connections for each device by right-clicking on the device and looking at the *"Connections*" tab in the *Device Properties Window*.

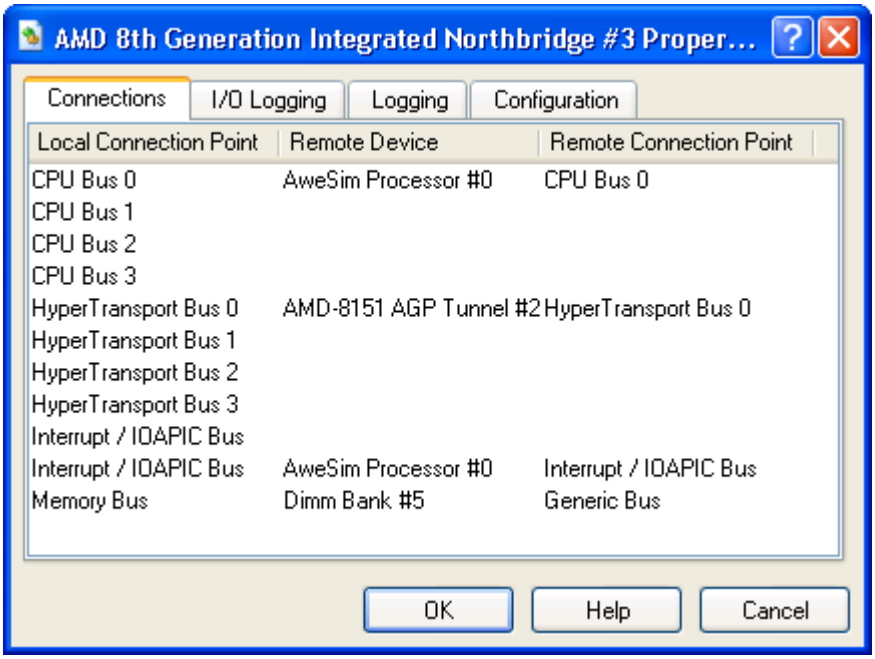


Figure 6-2: Connections Tab of Device Properties Window

8. Add the DIMM Device. Connect it to the AMD 8th Generation Integrated Northbridge, using the Northbridge's Memory Bus and the DIMM's Generic Bus.
9. Add the AMD-8151™ AGP Tunnel. This is a HyperTransport™ tunnel and AGP bridge. Connect it to the Northbridge using each device's HyperTransport Bus 0.
10. Add the Matrox Millenium G400 Graphics Device. This is the simulated video device. Connect it to the AMD-8151 AGP Tunnel Device using AMD-8151 AGP Tunnel AGP Bus and the Graphics Device's AGP or PCI Bus.

11. Add the Southbridge Device. Connect it to AMD-8151 AGP Tunnel using AMD-8151 AGP Tunnel HyperTransport Bus 1 and HyperTransport Bus 0. Also, connect AMD-8111™ to the DIMM device using AMD-8111 System Management Bus 0 and DIMM's Generic Bus.
12. Add the Winbond W83627HF SIO device. This is a Super IO device that supports keyboard, mouse, and floppy disk. Connect it to Southbridge using Winbond's Generic Bus and Southbridge's LPC Bus.
13. Add the PCI Bus. Connect it to AMD-8111 Southbridge using both devices' PCI Bus 0.
14. Add the Memory Device. This will contain the System BIOS image. Connect it to AMD-8111 Southbridge device using AMD-8111 LPC Bus and the Memory Device's Generic Bus.

6.3 Solo.bsd Device Configuration

To configure each device, right-click on the device and choose *Configure Device* from the workspace popup menu (see also Section 7, "*Device Configuration*", on page 53).

1. *Configure the Matrox Millenium G400 Graphics Device.*
   - Go to its *Configuration* tab.
   - Choose the BIOS file *Images/g400_897-21.bin*.
2. *Configure the Memory device.*
   - Go to its *Memory Configuration* tab.
   - Set the base address to fffc0000.
   - Set the *Size* to 8.
   - Set the Init File to *Images/ASLA00-3.BIN*.
   - Check the boxes for *Read Only*, *System BIOS ROM*, *Memory Address Masking*, *Memory is non-cacheable*.
   - Clear the boxes for "*Initialized unwritten memory*.
3. *Configure the PCI device.*
   - Go to its *PCI Bus Configuration* tab.
   - For the *PCI Slot 1*, add device ID 4, set *Base IRQ Pin* to *PCIIRQ A*, and check the *Enable Slot* box.
   - For the next three devices, use *Device IDs* 5, 6, and 7, with *PCIIRQs* B, C, and D, in that order. Check their "*Enable Slot*" boxes as well.

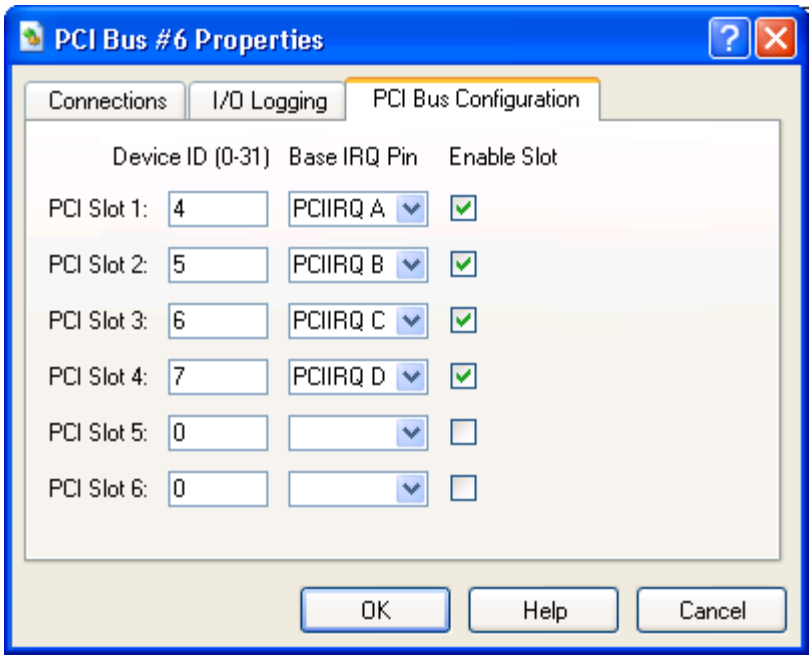


Figure 6-3: PCI Bus Configuration dialog window

4. *Configure the DIMM Memory device.*
   - Go to the *Dimm 0* tab.
   - Click *Import SPD*.
   - Open the SPD file *Images/simnow_DDR_256M.spd*.
5. *Configure the AweSim CPU device.*
   - Go to the *Processor Type* tab.
   - Choose the *Ahtlon64-754_SH-C0_(800MHz).id* product file, as shown in Figure 7-1 on page 56.

6.4 Save and Run

The created simulated computer is identical to the "s*olo.bsd*" computer. You can close the *Device Window* and save the file from the "*File→Save BSD*" or by clicking on the button. All that remains is to set up disk images (see Section 4.1, "*Creating A Blank Hard-Drive Image*", on page 35, Section 5.2.1, "*Assigning Disk-Images*", on page 42, and Section 13, "*DiskTool*", on page 161) and run the simulation.

7 Device Configuration

Each section in this chapter provides a description of how to configure device models in the simulator's *Device Properties* window. These device models include the CPU, CPU debugger, Northbridge, DIMM memory modules, AMD graphics device, Southbridge, Super IO, memory device, PCA9548- and PCA9556-SMB, PCI bus, AMD-8131™ PCI-X® device, PCI-X test device, AMD-8132™ PCI-X2 device, Raid device, SMB Hub device, EXDI server and the USB keyboard and mouse devices. These sections should be considered as a reference for how to configure a device model and are not intended to document how to use the model within the simulator.

The full release version of the simulator ships with more devices then the public release version. Table 7-1 gives an overview of supported devices depending on the simulators version.

Symbol	Device	Public Release	Full Release
	AMD Debugger	✔	✔
	AweSim Processor	✔	✔
	DIMM Device	✔	✔
	AMD 8th Generation Integrated Northbridge	✔	✔
	AMD-8111™ Southbridge	✔	✔
	AMD-8131™ PCI-X® Controller	✔	✔
	AMD-8132™ PCI-X Controller	✔	✔
	AMD-8151™ AGP Bridge Device	✔	✔
	Emerald Graphics Device	✔	✔
	Matrox® G400/G450 Graphics Device	✔	✘
	PCI Bus	✔	✔
	PCI-X Test Device	✘	✔
	Winbond W83627HF SIO	✔	✔
	Memory Device	✔	✔
	SMB Hub Device	✔	✔
	PCA9548 Device	✔	✔
	PCA9556 Device	✔	✔
	LTC4306 Device	✔	✔
	AT24C Device	✔	✔
	USB JumpDrive	✔	✔
	Desktop Network Adapter	✔	✔
	EXDI Server	✘	✔
	Compaq SmartArray 5304	✘	✔

Symbol	Device	Public Release	Full Release
	USB Keyboard	✗	✓
	USB Mouse	✗	✓
	XTR Device	✗	✓
	ITE 8712 SIO	✓	✓
	ATI SB600/RD790	✓	✓
	ATI SB700/RS780	✓	✓
	ATI SB400/RS480//RD890	✗	✓
	AMD “Istanbul”/AMD “Sao Paulo”/AMD “Magny-Cours”	✗	✓
	AMD “DeerHound” revB QuadCore Socket L1 device	✓	✓

Table 7-1: Supported Devices

To open a Device Property dialog window, open the Device View window “*View→Show Devices”* or click on the button. Then Open the workspace popup menu, right-click on a device in the workspace area and select “*Configure Device*”.

7.1 AweSim Processor Device

The AweSim processor device provides a simulation of an AMD microprocessor.

Interfaces
Three interfaces are used in the AweSim device:

CPU Bus 0. This interface is used to issue memory and I/O read and write requests, as well as cache control and input/output signal messages. This interface is generally connected to the Northbridge device.

Interrupt Bus. This interface is used to communicate interrupt request and acknowledge messages. This interface is connected to whichever device is used to generate and control interrupts - typically the Southbridge device.

System Messages Interface. This interface is used by the processor device to output ASCII and binary log information.

Initialization and Reset State
The processor device's state at initialization is equivalent to an industry-standard x86 processor at initialization. The L1 cache and APIC interfaces are disabled, the debugger is off, and the L1 cache is configured as two 2-way, 512-line, and 64-byte caches.

When the processor device receives a reset, the device resets its internal state in a manner consistent with a standard x86 processor. No configuration information is modified.

Contents of a BSD
The BSD file contains the current state of all internal processor registers, state variables, etc. It also contains all configuration information. Any memory configured locally to the processor is saved in the BSD.

Configuration Options
The *Device Properties Window* is used to set various processor identification and behavior options. Figure 7-1 shows the *Processor Type* tab for the AweSim processor device. Here you can specify which member of the AMD microprocessor family should be simulated. The default is a standard AMD microprocessor. See Section A.2.3, *Product Files (*.ID)*, on page 178.

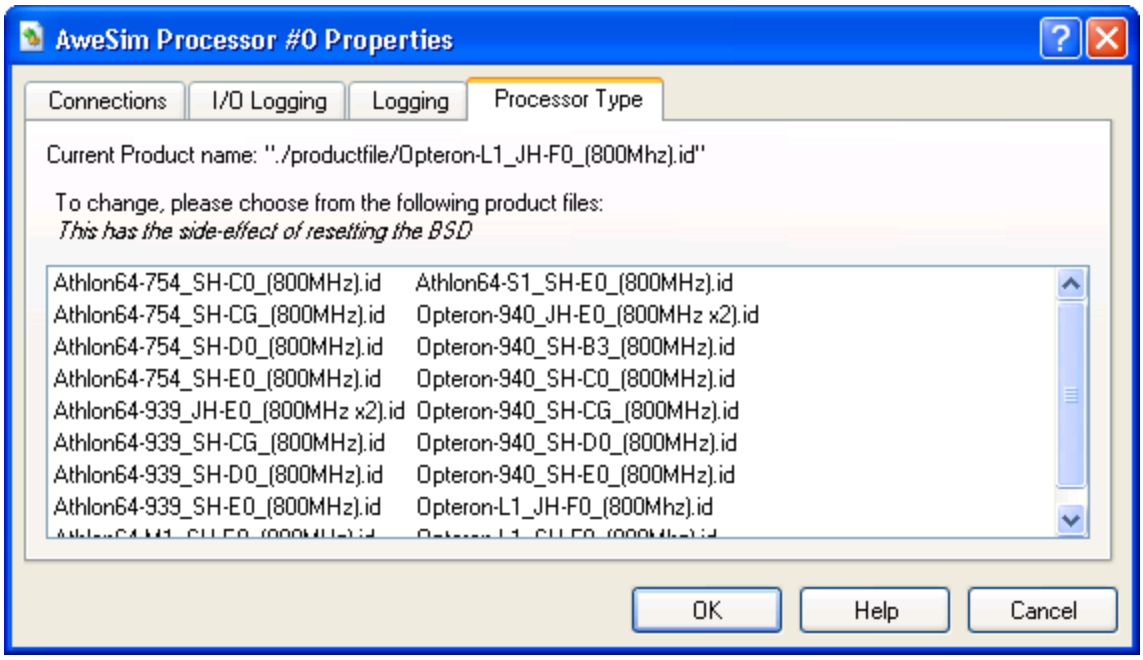


Figure 7-1: AweSim Processor-Type Properties

Figure 7-2 shows the *Logging* tab for the AweSim processor device. Here you can specify the following configuration options:

Check the *Log Disassembly* check box to log the disassembly of the instructions executed by the processor model.

Check the *Log Register State Changes* check box to log all the processor model register state changes.

Check the *Log I/O Read/Writes* check box to log all real I/O (not memory I/O) generated by the processor model.

Check the *Log Linear Memory Accesses* check box to log all memory accesses based on linear memory. This logs all 'data' memory accesses generated by the processor model. This does not log code fetch memory accesses, nor “physical” memory accesses (for example, page table access-and dirty-bit updates).

Check the *Log Exceptions* check box to log all exceptions generated by the processor model.

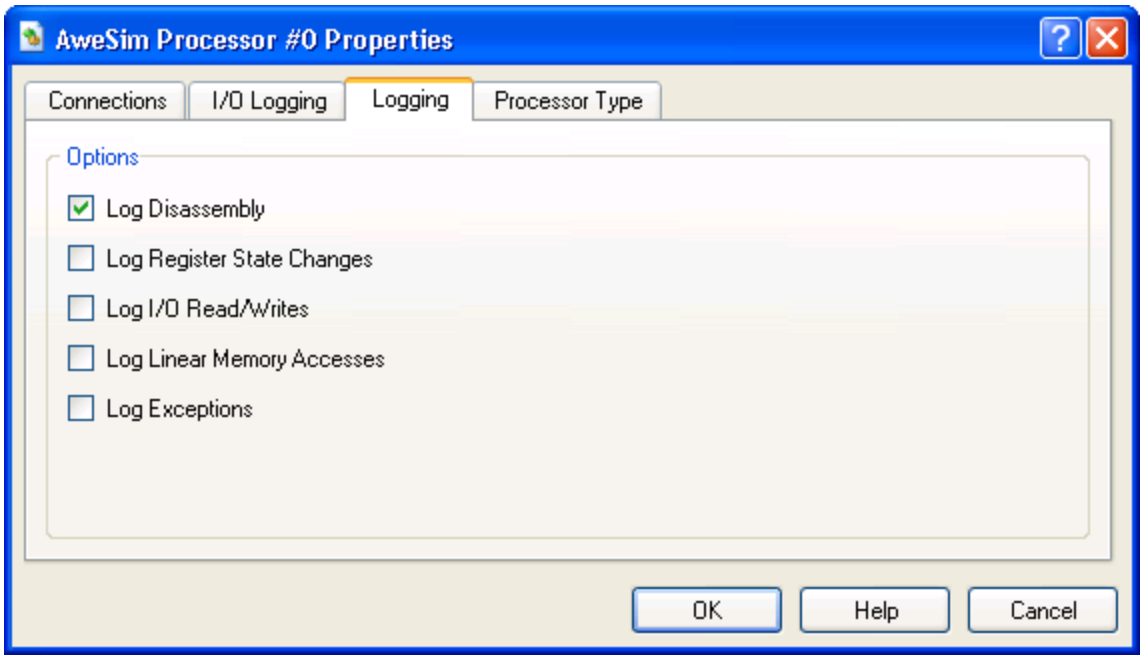


Figure 7-2: AweSim Processor Logging Properties Dialog

Log Messages

This device produces log messages to the Message Log Window as specified by the options in the Message Log Windows (see Section 9 "*Logging*", on page 141).

Difference from Real Hardware

While the processor device is a faithful simulation of the software-visible portion of an AMD microprocessor, it is not a model of the specific AMD microprocessor hardware. Because of this, the processor device is not equivalent in certain areas. Any issues related to timing, such as the time to execute a given instruction, will be different. The TLB models do not exactly match any particular processor, so any software that depends on exact TLB walking behaviors may not function correctly.

7.2 Debugger Device

The debugger allows debugging tasks such as break-pointing, single-stepping, and other standard tasks.

Interfaces
The debugger has no interfaces; the debugger is present if it is in the *Device Window*. To add the Debugger Device follow these steps:

1. Select "*View→Show Devices*".
2. Click and drag the Debugger Device icon from the device list on the left side into the workspace area on the right side of the Device Window.
3. Add an additional debugger for each processor you wish to debug.

Initialization and Reset State
The debugger initially is disabled and attached to processor 0.

Configuration Options
In the *Main Window*, select "*View→Show Debugger*". Click the Attach button to configure which processor is being debugged.

To use the CPU Debugger, please refer to Section 10.1, "*Using the CPU Debugger*", on page 147.

Log Messages
This device does not create log messages.

7.3 DIMM Device

The DIMM device provides a simulation model of an array of up to four dual-inline-memory modules (DIMMs). The model provides RAM storage and serial presence detect (SPD) ROM access for each DIMM. Bytes 0, 5, 13, and 31 (zero-based) of the SPD data are used to configure the DIMM model. The remaining SPD entries are available for BIOS probing, but are not used to configure the DIMM model.

The RAM array for each DIMM is sized based on parameters contained in the SPD array. SPD array bytes 5 and 31 are used to calculate the size of the DIMM's RAM array. If byte 0 in the SPD array has a value of zero, then the DIMM device does not respond to any SMBUS read attempts on the module. This indicates to the reading device that an SPD ROM is not available on the DIMM module. By appropriately setting bytes 5 and 31, and clearing byte 0, the model simulates a valid DIMM that contains no SPD ROM.

Dual data rate (DDR) DIMMs use bidirectional data strobe signals to latch data on transfers. The Northbridge device contains Programmable Delay Lines (PDLs) that are used to delay the Data Qualification Signal (DQS) signals so that the edges are centered on the valid data window. BIOS algorithms are used to locate the valid data window and adjust the PDLs accordingly.

Physical DIMMs provide 8 bytes of data per access. On the module, the 8 bytes of data are stored across several memory devices. The data width of the memory devices on the DIMM (SPD byte 13) determines how many PDLs are used. DIMMs that use 8-bit or 16-bit memory devices use one PDL per byte of width (eight total PDLs). DIMMs that use 4-bit devices use one PDL per nibble (16 total PDLs).

The memory controller in the AMD Opteron™ processor includes two DDR channels that are ganged into a single effective 128-bit interface. Each access to memory hits a pair of 64-bit DIMMs, where one DIMM supplies the lower 64 bits while the other DIMM supplies the upper 64 bits. Each DIMM must have the same arrangement in size and number of banks.

For each valid access to DRAM, the memory controller will assert one of eight bank-select lines (CS7:0). Each bank-select line selects one “virtual bank.” A virtual bank is the combination of one bank on the lower DIMM, and the corresponding bank on the upper DIMM. Row and column addresses select the data offset within the virtual bank.

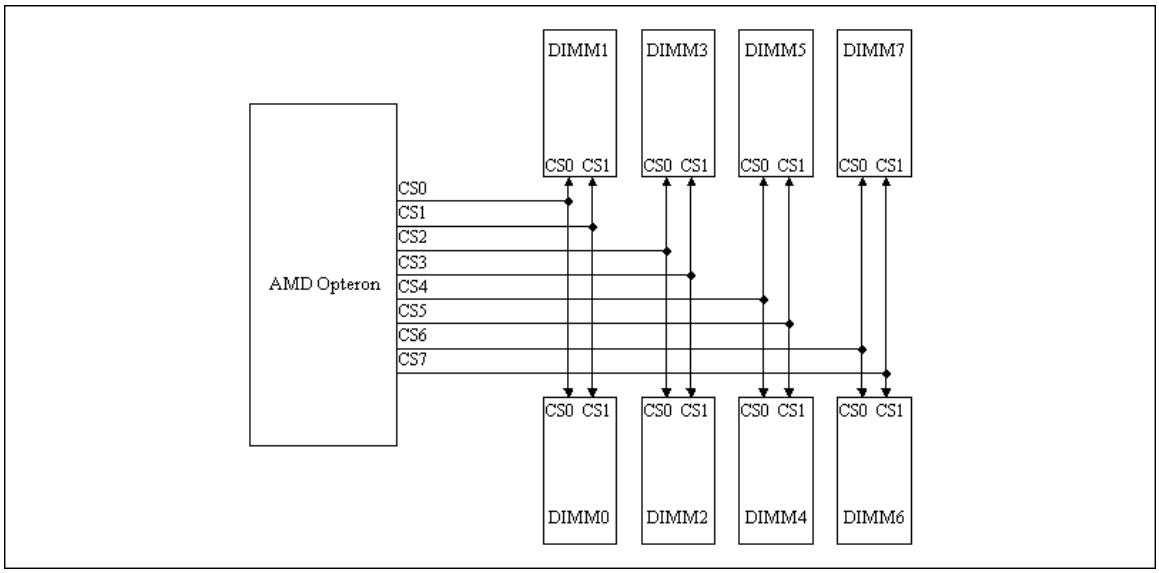


Figure 7-3: AMD Opteron™ Processor Virtual Bank-Select Line Configuration

Memory controllers in AMD Athlon™ 64 provide eight bank select lines. However, in this case, each bank-select is routed to only one physical DIMM bank, i.e., the banks are not ganged.

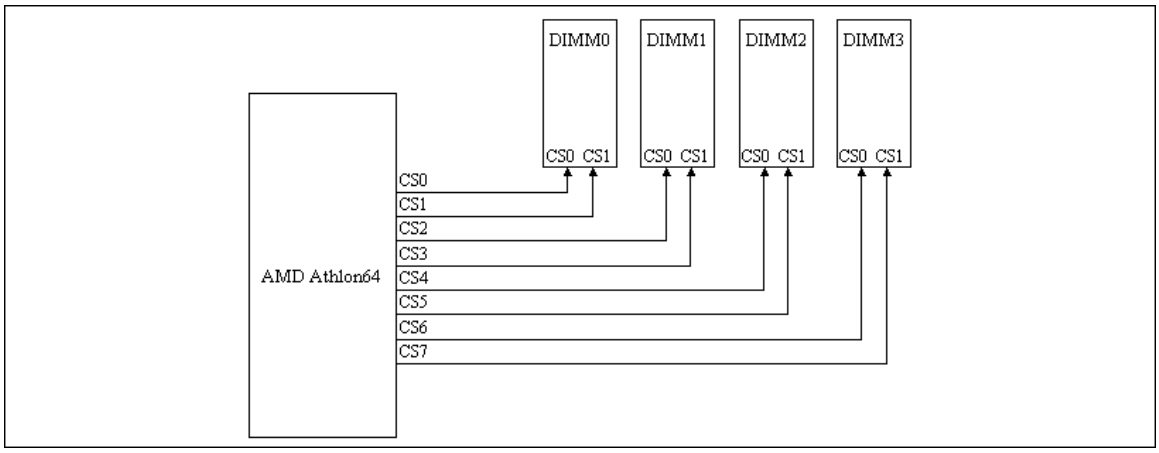


Figure 7-4: AMD Athlon™ 64 Processor Bank-Select Line Configuration

Configuration of the DIMM Device allows the user to specify SPD data for each simulated DIMM. The number of DIMMs supported in the DIMM Device model is dependent on the type of CPU used in the system. If the CPU type is an AMD Opteron processor, then the DIMM Device will assume a 128-bit memory interface and therefore allow configuration of up to eight individual DIMMs. If the CPU type is something other than AMD Opteron, then the DIMM device assumes a 64-bit memory interface and accepts configuration for only four DIMMs. It isn't until the simulation is started that the DIMM Device can determine what type of CPU is present. For this reason, the DIMM Device will initially display configuration tabs for 8 DIMMs even when used with a CPU that is not based on the AMD Opteron processor. After the simulation is started, the DIMM device will remove and ignore any configuration of DIMMs 4-7 if a processor other than the AMD Opteron is detected.

Once the simulation is started, the DIMM Device allocates memory arrays to hold the DRAM data. One array is allocated for each bank or virtual bank. In the case of 64-bit memory interfaces, memory arrays are allocated to match the size of the physical banks on each DIMM. If the memory interface is 128 bits, then the memory arrays are sized to the sum of the physical bank pairs that make up the virtual banks. For example:

Virtual bank0 is the combination of physical bank0 on DIMM0 and physical bank0 on DIMM1. If physical bank0 on each DIMM is 32MB in size, then the array allocated for virtual bank0 is sized at 64MB.

Each virtual bank is handled like it is one large bank, rather than two combined smaller banks. The model does not distinguish between addresses that hit in the upper physical bank and addresses that hit in the lower physical bank.

Memory read- and write-messages sent to the DIMM Device use the same structure for both 128-bit and 64-bit interfaces. Each message includes a bank select field, an address field, and a data size field. The bank select field implements the CS7:0 lines while the address field specifies the beginning offset within the bank/virtual bank, and the data size field specifies the size of the datum.

Interfaces
The DIMM device is implemented as a single-interface device. However, the device accepts two distinct classes of messages: RAM read/write messages, and SMBUS reads of SPD data. In most system configurations, the DIMM device is connected to a Northbridge device's DIMM interface as well as a Southbridge device's SMBUS interface.

Initialization/Reset State
On creation of the DIMM device, all RAM arrays are set to all ones, and SPD ROM arrays are cleared. Reset initializes the RAM arrays to all ones, but does not alter the SPD ROM arrays. Configuration options are not affected by reset.

Contents of a BSD
The RAM arrays, SPD ROM arrays, and all configuration option settings are saved in the BSD.

Configuration Options

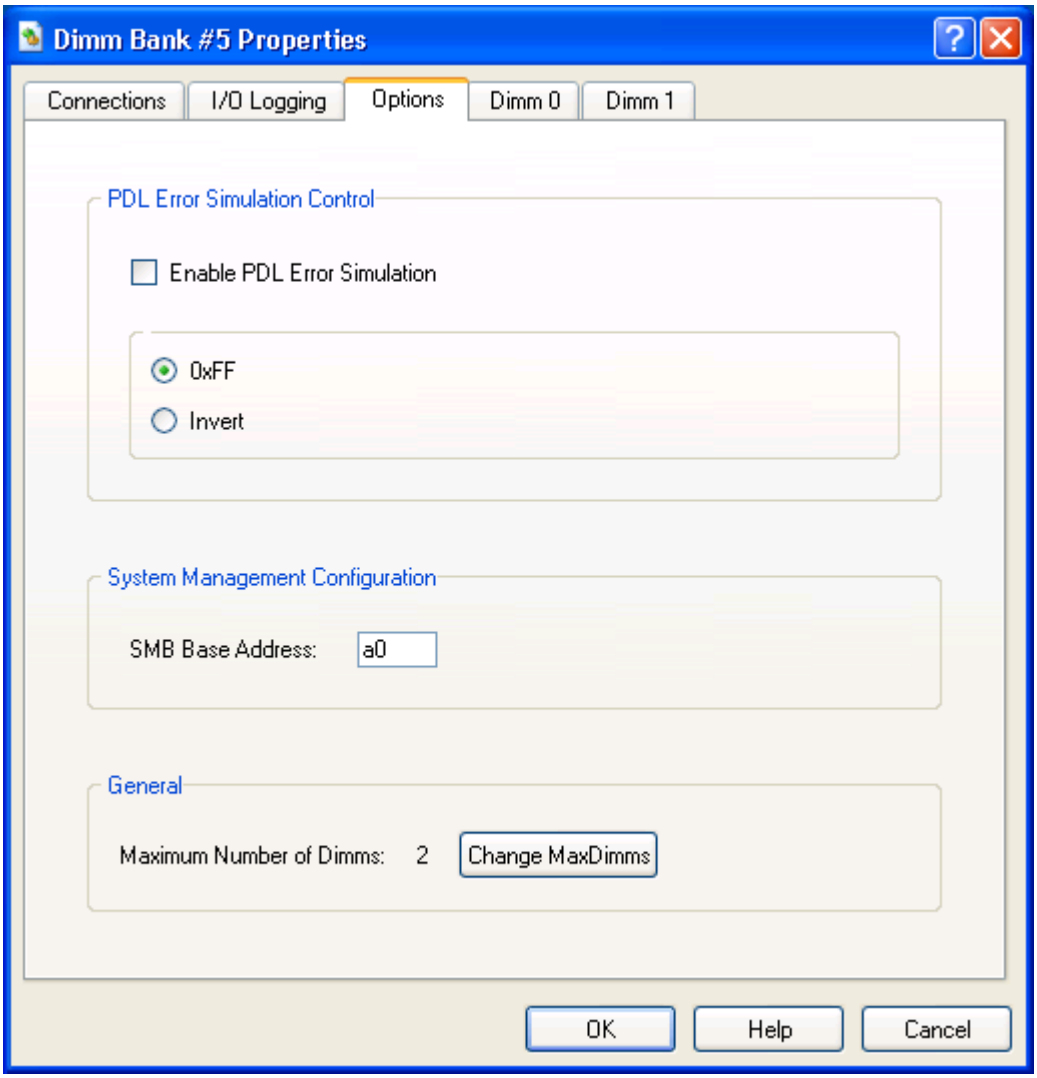


Figure 7-5: DIMM-Bank Options Properties Dialog

Figure 7-5 shows the dialog for configuring DIMM-bank options.

The *PDL Error Simulation Control* section specifies the type of error that the DIMM device will generate, when a memory read is attempted and when a Northbridge PDL is set outside the valid response range. These settings apply to all four simulated DIMMs.

If *Enable PDL Error Simulation* is selected, then the DIMM device monitors PDL settings for all RAM reads. The *0xFF* option specifies that the return data should be forced to all ones. The *Invert* option specifies that the return data should be a bitwise inversion of the valid data.

The *SMB Base Address* entry selects the 8-bit address that this DIMM device responds to. The SMB address is used for the reading of DIMM SPD data

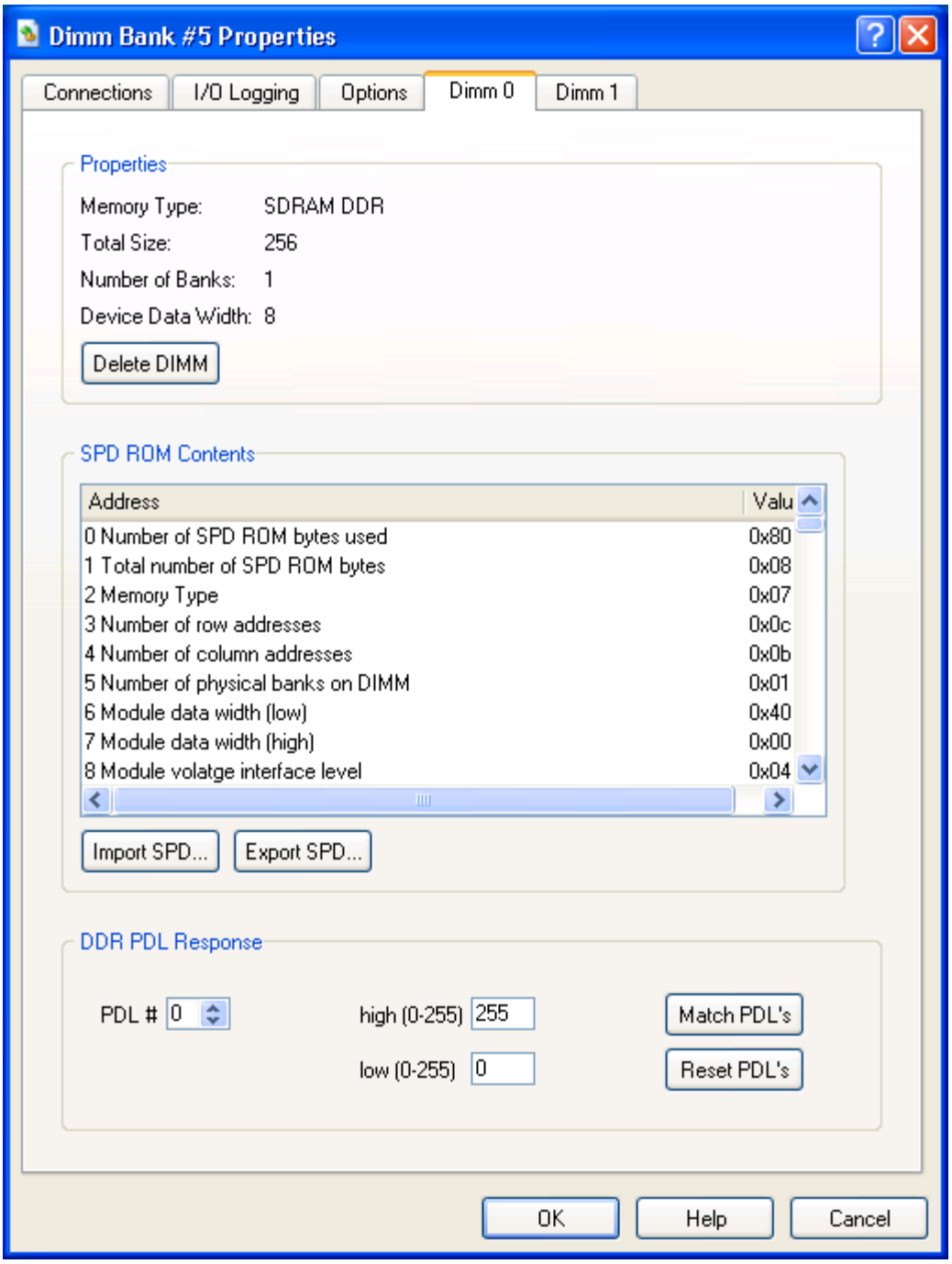


Figure 7-6: DIMM Module Properties Dialog

The two DIMM module configuration dialogs, shown in Figure 7-6, (DIMM0 – DIMM1) provide module-specific setup options for each simulated DIMM. The two DIMM module configuration dialogs share the same format.

Note: The public release of the simulator does not support any of the options shown in Figure 7-6. To change the simulated memory size please use the Memory Configurator, see Section 14.2, "Changing DRAM Size", on page 167.

The upper part of the dialog lists some summary information. This information, which is derived from the SPD data, gives a quick indication of the type of device being simulated.

The center section of the dialog lists all 256 bytes of data held in the simulated SPD ROM. The list box provides a description of each byte index in the ROM. If a description is selected, the corresponding data byte is displayed in the text box to the right.

The *Import SPD* and *Export SPD* buttons provide the option of loading and saving SPD ROM data. The file format is an unformatted binary image, with an extension of "*.*spd"*.

The bottom section of the dialog is used to configure *DDR PDL Response* ranges for the simulated DIMM. PDL response ranges can be individually set for each of 16 PDLs. Adjusting the *Low* and *High* value modifies the response range for a particular PDL. When an appropriate response range is set for one PDL, the same range can be applied to all 16 PDLs by clicking on the *Match PDLs* button. The *Reset PDLs* button sets all 16 PDL response ranges to their maximum range (0 - 255).

Log Messages
This device does not produce log messages.

Difference from Real Hardware
The DIMM device does not simulate timing-related issues except for PDL error simulation. The performance of real DIMM hardware is highly dependent on timing and loading issues.

ECC simulation is not provided.

7.4 Emerald Graphics Device

The Emerald graphics device provides an industry-standard PCI/AGP VGA-compatible video device. The device provides a fully functional set of PCI configuration registers. The AGP interface is currently somewhat minimal, and is not capable of generating AGP cycles nor AGP-specific modes at this time.

The Emerald graphics device is comprised of a standard VGA and the Emerald Graphics sub device. The graphics display engine automatically switches between the Emerald Graphics sub device and the VGA as necessary to display the selected video modes, with only one being able to display at a time. The VGA sub device provides an industry-standard VGA interface used by BIOS and DOS. The Emerald Graphics device provides an AGP and PCI graphics device interface controllable either by VESA BIOS extensions or a video driver. In addition to the VGA standard modes, Emerald Graphics supports a wide range of graphics modes from 320x200 at 16-bit color up to 2048x1536 at 32-bit color with either the VESA BIOS extensions or a video driver.

Interfaces
The Emerald graphics device has both a PCI slot and an AGP bus connection, only one of which can be used at any time to connect to PCI slots or AGP bus ports in other devices.

Initialization and Reset State
Upon initial creation, this device initializes the internal registers to VGA standard reset state, and creates a display window that acts as the VGA display. The Configuration options are initialized to enable both the VGA and Emerald Graphics. The frame-buffer size is initialized to 16 Mbytes and the Bios File memory area is initialized to all ones.

A reset will re-load the default PCI configuration registers and place default values in the Chip and FIFO configuration for the Emerald Graphics device.

Contents of a BSD
The data saved in the BSD depends on the mode the graphics controller was in when the BSD was saved. If the graphics controller was in VGA mode, the BSD file contains the contents of all VGA registers, a copy of the 256-Kbyte VGA frame buffer, and all configuration information. If the graphics controller was in a high-resolution mode (non-VGA in Windows) the frame buffer, Emerald Graphics registers, and PCI configuration registers are saved in the BSD. When the BSD file is reloaded, all registers and the frame buffer are restored, and a display image is captured and displayed in the display window.

Configuration Options

VGA Sub Device Configuration

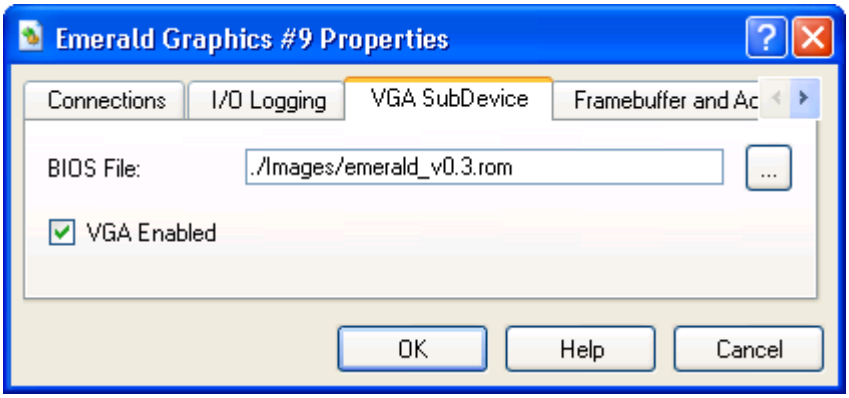


Figure 7-7: Graphics-Device VGA Sub Device Properties Dialog

In Figure 7-7, the *BIOS File* option enables you to load different VGA BIOS ROMs into the device. The VGA ROM is assumed to be a maximum of 32-Kbytes, and is assigned to ISA bus address 0x000C0000 - 0x000C7FFF, which is the industry-standard location. This file must be a standard binary file, with the correct header and checksum information already incorporated.

The VGA enabled checkbox enables or disables the VGA registers. If it is not checked, the VGA registers are not updated and the display window will not display from the VGA frame buffer.

Frame Buffer Sub Device Configuration

In Figure 7-8, the *Frame Buffer Size (Mbytes)* sets the size of the frame-buffer in megabytes. The value placed in this option is only read at reset. The frame-buffer size can not be dynamically modified.

The *Accelerator Enabled* checkbox enables or disables the graphics accelerator. The accelerator is enabled by default.

The *VESA BIOS Extensions Enabled* checkbox enables or disables the VESA BIOS support. The VESA BIOS Extensions are enabled by default.

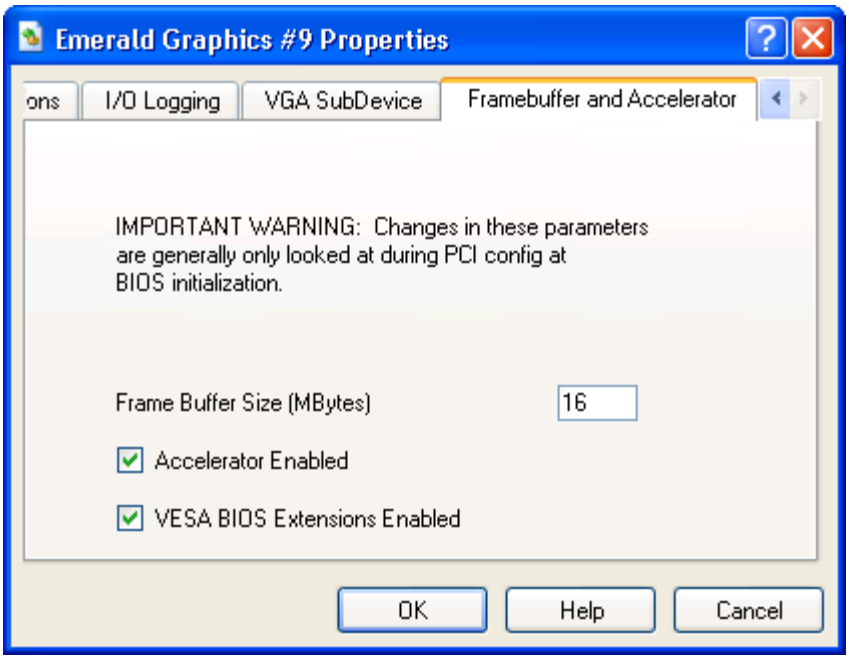


Figure 7-8: Graphics Device Frame Buffer SubDevice Properties

Difference from Real Hardware

The Emerald Graphics device currently does not simulate any specific graphics hardware, it simulates something functionally "like" a modern graphics adapter, with only 2D acceleration implemented at this time. Drivers are Windows only at the moment.

When the VGA display window has the focus, any keyboard messages and mouse-click messages received by the window are routed via a *DEVCWINDOWMSG* message through the simulators I/O subsystem. The keyboard or mouse device accepts these messages and simulates key-presses and key-releases to match the keys. While certain key combinations do not result in the generation of keyboard messages by the OS, this does enable you to use the real keyboard to interact with the simulation in many cases.

Supported VESA BIOS Graphics Modes

Only supports flat and linear frame buffer, with 16-bit/64K (5:6:5) colors and 32-bit/16.8M (8:8:8:8) colors modes.

Table 7-2 shows the subset of "standard" VESA mode numbers supported.

Mode Number	Resolution	Color depth
10Eh	320x200	16-bit
111h	640x480	16-bit
114h	800x600	16-bit
117h	1024x768	16-bit
11Ah	1280x1024	16-bit

Table 7-2: Supported Standard VESA Modes

Table 7-3 shows the supported custom VESA mode numbers.

Mode Number	Resolution	Color depth
140h	320x200	32-bit
141h	640x480	32-bit
142h	800x600	32-bit
143h	1024x768	32-bit
144h	1280x720	16-bit
145h	1280x720	32-bit
146h	1280x960	16-bit
147h	1280x960	32-bit
148h	1280x1024	32-bit
149h	1600x1200	16-bit
14Ah	1600x1200	32-bit
14Bh	1920x1080	16-bit
14Ch	1920x1080	32-bit
14Dh	1920x1200	16-bit
14Eh	1920x1200	32-bit
14Fh	2048x1536	16-bit
150h	2048x1536	32-bit

Table 7-3: Supported Custom VESA Modes

Improve Graphics Performance

When you run Windows in simulation and you open a menu, list box, tool-tips, or other screen element, the object may open very slow. To disable this option, use the following steps:

1. Click **Start**, point to Settings, and then click **Control Panel**.
2. Double-click **Display**.
3. Click **Effects**, clear the **Use the following transition effects for menus and tooltips** check box, click **ok**, and then close Control Panel.

7.5 Matrox MGA-G400 PCI/AGP

The Matrox G400 graphics device provides a high performance PCI/AGP VGA-compatible video device. The device provides a fully functional set of PCI configuration registers, and a 2D drawing engine. The AGP interface is currently somewhat minimal, and is not capable of generating neither AGP cycles nor AGP-specific modes at this time.

High performance device drivers are available for most operating systems (Windows, Linux, and Solaris). The Matrox G400 supports full acceleration of all GDI and DirectDraw functions.

Figure 7-9 shows the integrated components of the Matrox G400 graphics device. Features and components which are currently not supported by the Matrox G400 graphics device model have a ⃠ symbol in the following block diagram.

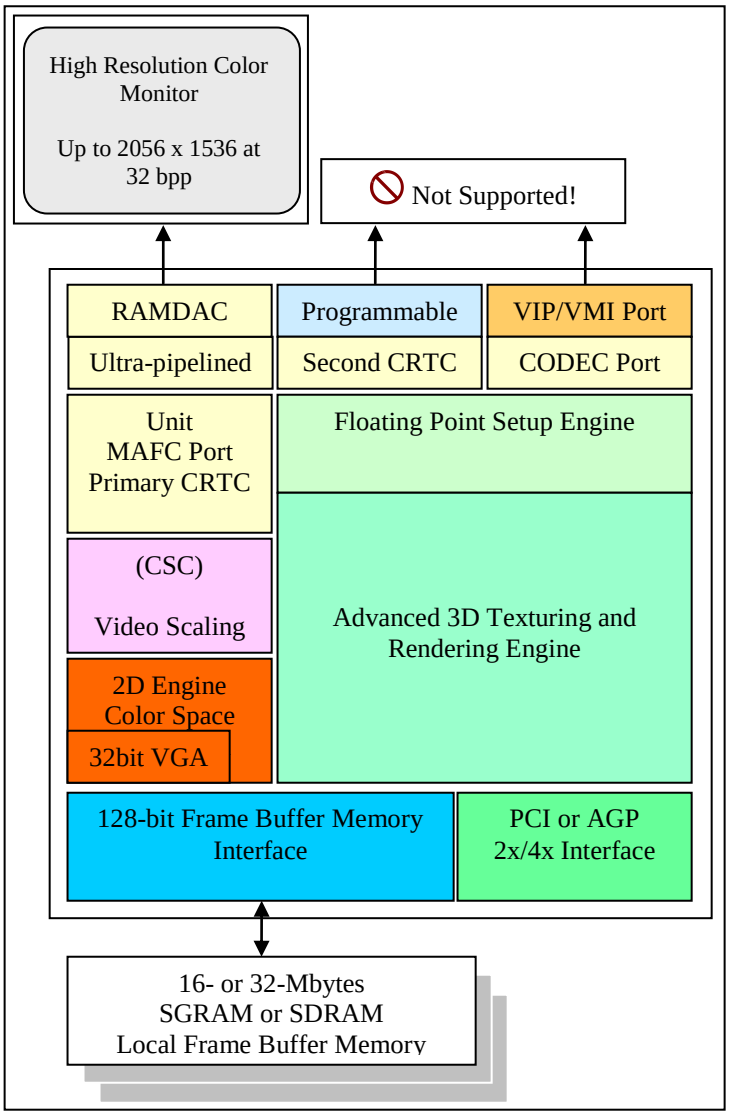


Figure 7-9: Matrox G400 Block Diagram

Interfaces
The Matrox G400 graphics device has both a PCI bus and an AGP bus connection, only one of which can be used at any time to connect to PCI bus or AGP bus ports in other devices.

Initialization and Reset State
Upon initial creation, this device initializes the internal registers to Matrox G400 standard reset state, and creates a display window that acts as the VGA display. The Configuration options are initialized to enable both the VGA and Matrox Power Graphics Mode. The frame-buffer size is initialized to 32 Mbytes and the Bios File memory area is initialized to all ones.

A reset will re-load the default PCI configuration registers and place default values in the Chip and FIFO configuration for the Matrox G400 graphics device.

Contents of a BSD
The data saved in the BSD depends on the mode the graphics controller was in when the BSD was saved. If the graphics controller was in VGA mode, the BSD file contains the contents of all VGA registers, a copy of the 256-Kbyte VGA frame buffer, and all configuration information. If the graphics controller was in Matrox Power Graphics Mode (non-VGA in Windows) the linear frame buffer, Power Graphics registers, and PCI configuration registers are saved in the BSD. When the BSD file is reloaded, all registers and the frame buffer are restored, and a display image is captured and displayed in the display window.

Configuration Options
Figure 7-10 shows the *Information* tab. The following information describes the active configuration of the Matrox G400 graphics device.

The Graphics Hardware *Model* can be set to one of the following models:

- Matrox Millennium G400 PCI
- Matrox Millennium G400 AGP

Currently there is only support for the Matrox G400 chip with SingleHead feature support available.

The *Graphics BIOS* version is the version of the BIOS that is assigned and used by the graphics device. If you flash the BIOS the version number will change. For more information about flashing the graphics device BIOS see Figure 7-11.

The *Graphics Memory* section shows information about the current memory configuration of the graphics device. Currently supported memory configurations are:

- 32/16 MB SGRAM with 300 MHz RAMDAC
- 32/16 MB SDRAM with 300 MHz RAMDAC

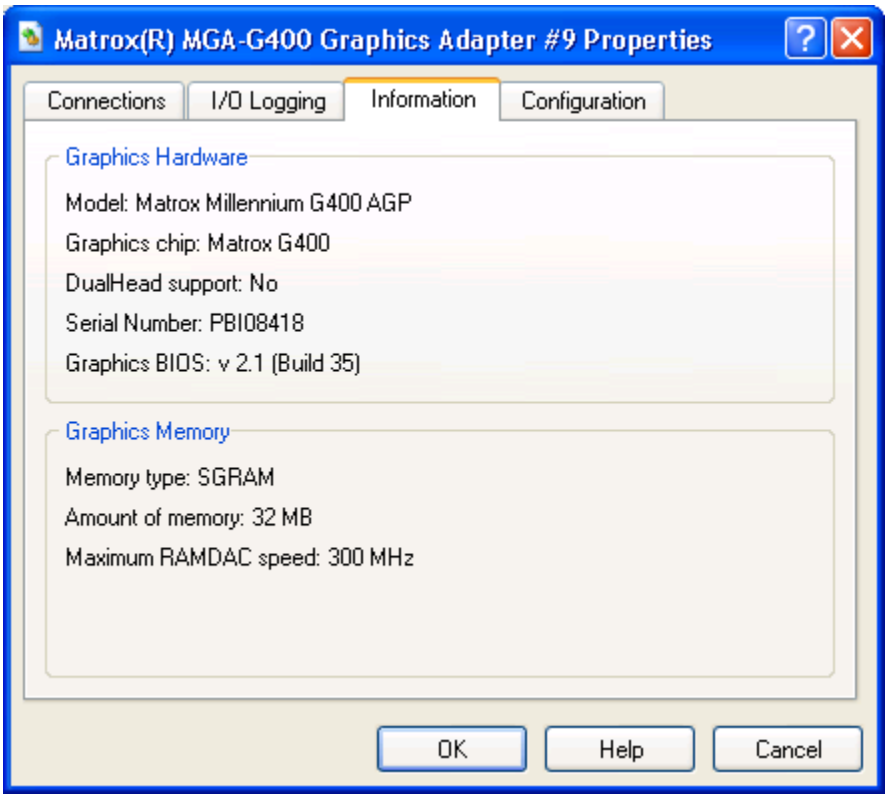


Figure 7-10: Matrox G400 Information Property Dialog

The *Configuration* tab displays details about the active configuration of the Matrox G400 graphics device.

If you want to change the active configuration, click on the *Configuration* Tab (see Figure 7-11).

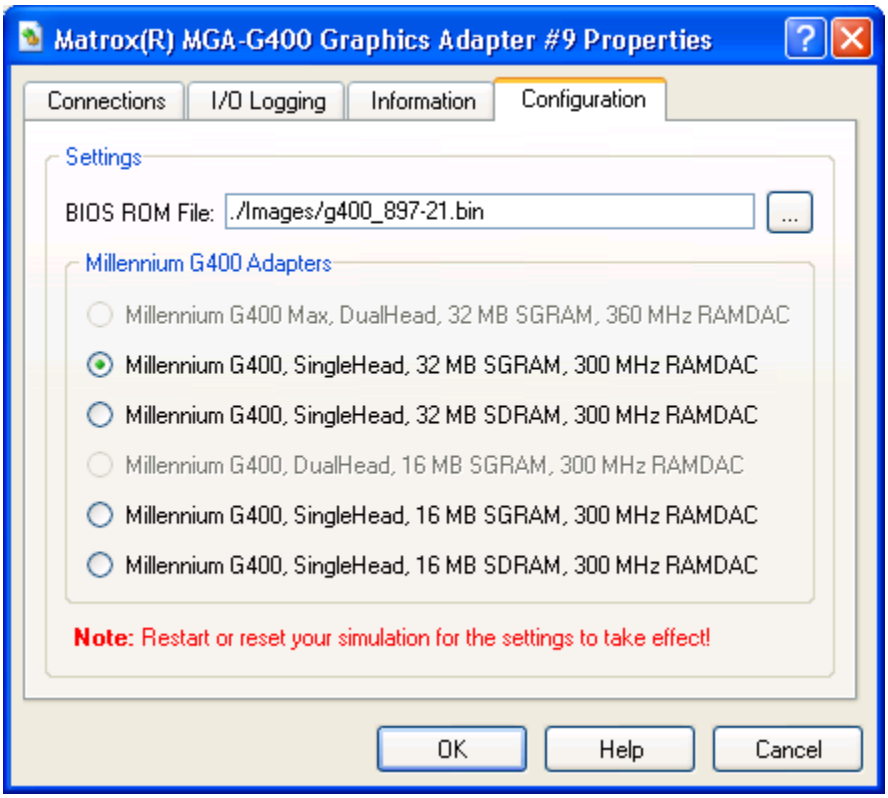

Matrox(R) MGA-G400 Graphics Adapter #9 Properties
Connections
I/O Logging
Information
Configuration
Settings
BIOS ROM File:
./Images/g400_897-21.bin
...
Millennium G400 Adapters
Millennium G400 Max, DualHead, 32 MB SGRAM, 360 MHz RAMDAC
Millennium G400, SingleHead, 32 MB SGRAM, 300 MHz RAMDAC
Millennium G400, SingleHead, 32 MB SDRAM, 300 MHz RAMDAC
Millennium G400, DualHead, 16 MB SGRAM, 300 MHz RAMDAC
Millennium G400, SingleHead, 16 MB SGRAM, 300 MHz RAMDAC
Millennium G400, SingleHead, 16 MB SDRAM, 300 MHz RAMDAC
Note: Restart or reset your simulation for the settings to take effect!
OK
Help
Cancel

Figure 7-11: Matrox G400 Configuration Properties

The *BIOS ROM File* input field gives you the ability to load different Matrox G400 BIOS ROMs into the device. This is in particular useful if Matrox releases a new BIOS ROM file which has improvements or bug fixes.

To check for new Matrox BIOS ROM releases go to http://www.matrox.com/mga/support/drivers/bios/.

The Matrox G400 ROM has a maximum size of 32-Kbytes, and is assigned to ISA bus address 0x000C0000 - 0x000C7FFF, which is the industry-standard location.

The *Configuration* tab lets you choose from six different Matrox G400 graphics adapters. For instance, if you prefer to use a Matrox Millennium G400, SingleHead, 16 Mbytes of SDRAM, with a 300 MHz RAMDAC, instead of the default adapter then select this adapter from the *Millennium G400 Adapters* list. To apply the new configuration, click on the 'Ok' button.

Note if you make any changes in the *Configuration* tab you **must** restart or reset your simulation before the new configuration will take effect!

Difference from Real Hardware

The Matrox G400 graphics device is a faithful simulation of the software-visible portion of a Matrox G400 adapter; it is not a model of the specific Matrox G400 hardware. Because of this, the graphics device is not equivalent in certain areas. Any issues related

to timing, such as the vertical retrace time, will be different. Any software that depends on exact timing behavior may not function correctly.

The following features are only partially implemented. Any software that depends on these features may not function correctly.

- Translucency / Full Alpha-Blending
- Full Texture Mapping
- Gouraud Shaded Fills (ALPHA, FOG, STENCIL)
- Trapezoids functions
- Bitblts
  a. Color Patterning 8x8
  b. Expansion (Character Drawing) 1 bpp Planar
- Lines
  a. With Line-style
  b. With Depth
  c. Polyline/Polysegment using Vector Pseudo-DMA Mode
- Image Load (ILOAD)
  a. Linear-Color Expansion (Character Drawing) 1 bpp
  b. Loading the Texture Color Palette
- Loading any accelerator registers through the Pseudo DMA Window
- ZBuffer Direct Access Procedure when ZBuffer is in AGP Space
- Table-Fog
- Video Scaler
- Texture Unit blending
- Texture Staging

Supported 2D Features

- Bus-Mastering (PCI/AGP)
- Raster Operations: 0, ~(D | S), D & S, D & ~S, ~S, (~D) & S, ~D, D ^ S, ~(D & S), D & S, ~(D ^ S), D, D | ~S, S, (~D) | S, D | S, 1
- Hardware Clipping
- Software-/Hardware-Cursor
  a. Three-Color Cursor
  b. XGA Cursor
  c. X-Windows Cursor
  d. 16-Color Palletized Cursor
- Bitblts
  a. Two-Operand
  b. Transparent Two-Operand
  c. With Expansion (Character Drawing) 1bpp
- Image Load (ILOAD)
  a. Two-operand
  b. With Expansion (Character Drawing) 1bpp
- Rectangles

a. Patterned Fills
b. Constant Shaded
c. Gouraud Shaded (partially)
d. Texture Mapping (partially)

- Trapezoids
  a. Constant Shaded
- Lines
  a. Auto-Lines (line open/line close)
  b. Solid-Lines (line open/line close)
- 8, 15, 16, 24, and 32 Bits Per Pixel video modes
- ILOAD Pseudo- DMA Window Transfers
- Programmable, transparent BLTer
- Linear packed pixel frame buffer

Supported DirectX 6.1 Features

- Alpha Test0
- Alpha Blending Functions
  a. Normal-Blending
  b. Transparency-Blending
  c. Additive-Blending
  d. Soft-Additive-Blending
  e. Multiplicative-Blending
- Depth Test (Z-Buffer) 15-bit, 16-bit, 24-bit, and 32-bit
- Texel-Width (4-, 8-, 12-, 15-, 16-, and 32-bit
- UV Texture Coordinate support
- DMA-Vertex Engine

Supported Graphics Modes

The Matrox G400 provides three different display modes: text (VGA or SVGA), VGA graphics, and SVGA graphics. Table 7-4 list all of the display modes which are available through BIOS calls.

Mode Number	Type	Organization	Resolution	No. of colors	Supported
0x00	VGA	40x25 Text	360x400	16	✔
0x01	VGA	40x25 Text	360x400	16	✔
0x02	VGA	80x25 Text	720x400	16	✔
0x03	VGA	80x25 Text	720x400	16	✔
0x04	VGA	Packed-pixel 2 bpp	320x200	4	✔
0x05	VGA	Packed-pixel 2 bpp	320x200	4	✔
0x06	VGA	Packed-pixel 1 bpp	640x200	2	✔
0x07	VGA	80x25 Text	720x400	2	✔
0x0D	VGA	Multi-plane 4 bpp	320x200	16	✔
0x0E	VGA	Multi-plane 4 bpp	640x200	16	✔
0x0F	VGA	Multi-plane 1 bpp	640x350	2	✔
0x10	VGA	Multi-plane 4 bpp	640x350	16	✔
0x11	VGA	Multi-plane 1 bpp	640x480	2	✔
0x12	VGA	Multi-plane 4 bpp	640x480	16	✔
0x13	VGA	Packed-pixel 8 bpp	320x200	256	✔

Mode Number	Type	Organization	Resolution	No. of colors	Supported
0x0108	VGA	80x60 Text	640x480	16	✗
0x0109	VGA	132x25 Text	1056x400	16	✗
0x010A	VGA	132x43 Text	1056x350	16	✓
0x010B	VGA	132x50 Text	1056x400	16	✗
0x010C	VGA	132x60 Text	1056x480	16	✗
0x0100	SVGA	Packed-pixel 8 bpp	640x400	256	✓
0x0101	SVGA	Packed-pixel 8 bpp	640x480	256	✓
0x0110	SVGA	Packed-pixel 16 bpp	640x480	32K	✓
0x0111	SVGA	Packed-pixel 16 bpp	640x480	64K	✓
0x0112	SVGA	Packed-pixel 16 bpp	640x480	16M	✓
0x0102	SVGA	Multi-plane 4 bpp	800x600	16	✗
0x0103	SVGA	Packed-pixel 8 bpp	800x600	256	✓
0x0113	SVGA	Packed-pixel 16 bpp	800x600	32K	✓
0x0114	SVGA	Packed-pixel 16 bpp	800x600	64K	✓
0x0115	SVGA	Packed-pixel 32 bpp	800x600	16M	✓
0x0105	SVGA	Packed-pixel 8 bpp	1024x768	256	✓
0x0116	SVGA	Packed-pixel 16 bpp	1024x768	32K	✓
0x0117	SVGA	Packed-pixel 16 bpp	1024x768	64K	✓
0x0118	SVGA	Packed-pixel 32 bpp	1024x768	16M	✓
0x0107	SVGA	Packed-pixel 8 bpp	1280x1024	256	✓
0x0119	SVGA	Packed-pixel 16 bpp	1280x1024	32K	✓
0x011A	SVGA	Packed-pixel 16 bpp	1280x1024	64K	✓
0x011B	SVGA	Packed-pixel 32 bpp	1280x1024	16M	✓
0x011C	SVGA	Packed-pixel 8 bpp	1600x1200	256	✓
0x011D	SVGA	Packed-pixel 16 bpp	1600x1200	32K	✓
0x011E	SVGA	Packed-pixel 16 bpp	1600x1200	64K	✓

Table 7-4: Matrox G400 VESA Modes

Memory Interface

The Matrox G400 supports a total of 32 megabytes of SGRAM/SDRAM memory comprised of one or two banks of 8, 16, or 32 Mbytes each.

In Power Graphics Mode, the resolution depends on the amount of available memory. Table 7-5 shows the memory configuration for each standard VESA resolution in pixel depth.

	Single Frame Buffer Mode				Single Z-Buffer					
	No Z				Z 16 bits			Z 32 bits		
Resolution	**8-bit**	**16-bit**	**24-bit**	**32-bit**	**8-bit**	**16-bit**	**32-bit**	**8-bit**	**16-bit**	**32-bit**
640x480	8M	8M	8M	8M	8M	8M	8M	8M	8M	8M
720x480	8M	8M	8M	8M	8M	8M	8M	8M	8M	8M
800x600	8M	8M	8M	8M	8M	8M	8M	8M	8M	8M
1024x768	8M	8M	8M	8M	8M	8M	8M	8M	8M	8M
1152x864	8M	8M	8M	8M	8M	8M	8M	8M	8M	8M
1280x1024	8M	8M	8M	8M	8M	8M	8M	8M	8M	10M
1600x1200	8M	8M	8M	8M	8M	8M	16M	16M	16M	16M
1920x1080	8M	8M	8M	8M	8M	8M	16M	16M	16M	16M
1800x1440	8M	8M	8M	16M	8M	16M	16M	16M	16M	16M
1920x1200	8M	8M	8M	8M	8M	8M	16M	16M	16M	16M
2048x1536	8M	8M	16M	16M	16M	16M	32M	16M	32M	32M

Table 7-5: Supported Resolutions in Power Graphics Mode

Supported Guest Operating Systems
Table 7-6 shows all operating systems which are tested and known to work with the Matrox G400 graphics device model:

Guest Operating System	Device Driver Version	Known Issues
MS-DOS	N/A	No known issues.
Windows 2000	5.93.009	No known issues.
Windows XP (32-bit/64-bit)	5.93.009/1.11.00.114SE	No known issues.
Windows Server 2003 (32-bit/64-bit))	5.93.009/1.11.00.114SE	No known issues.
Windows Vista Beta 2 Build 5308 (32-bit/64-bit)	N/A (VESA only)	No known issues.
Linux (32-bit/64-bit), RedHat/SuSE/SuSE Xen	Standard MGA Driver	No known issues.
Solaris 10 for AMD64	XF86 MGA Solaris	No known issues.

Table 7-6: Supported Guest Operating Systems

Improve Graphics Performance
When you run Windows in simulation and you open a menu, list box, tool-tips, or other screen element, the object may open slowly. To disable this option, use the following steps:

1. Click **Start**, point to Settings, and then click **Control Panel**.
2. Double-click **Display**.
3. Click **Effects**, clear the **Use the following transition effects for menus and tool tips** check box, click **ok**, and then close Control Panel.

Or:

1. Right click on **My Computer** and select **Properties**.
2. Click on **Advanced**, **Performance**, and then on **Settings….**
3. Select the **Adjust For Best Performance** option.
4. Click on **Apply**.

Also make sure you have installed the Matrox G400 graphics device drivers. You can download the latest Matrox Millennium G400 graphic device drivers for Windows and Linux at http://www.matrox.com/mga/support/drivers/latest/home.cfm.

Enabling Graphics Hardware Acceleration on Windows Server Operating Systems
Graphics Hardware Acceleration and DirectX are disabled by default on a Windows Server configuration to ensure maximum stability and uptime. But if you need to improve the graphics performance the following steps will guide you through on how you can enable hardware acceleration.

1. Right-click the desktop, and then click **Properties** on the menu.
2. Click the **Settings** tab, and then click on **Advanced**.
3. Click the **Troubleshoot** tab.
4. Move the **Hardware Acceleration** slider across to **full** (see Figure 7-12).
5. Click **Ok**, and then click **Close**.

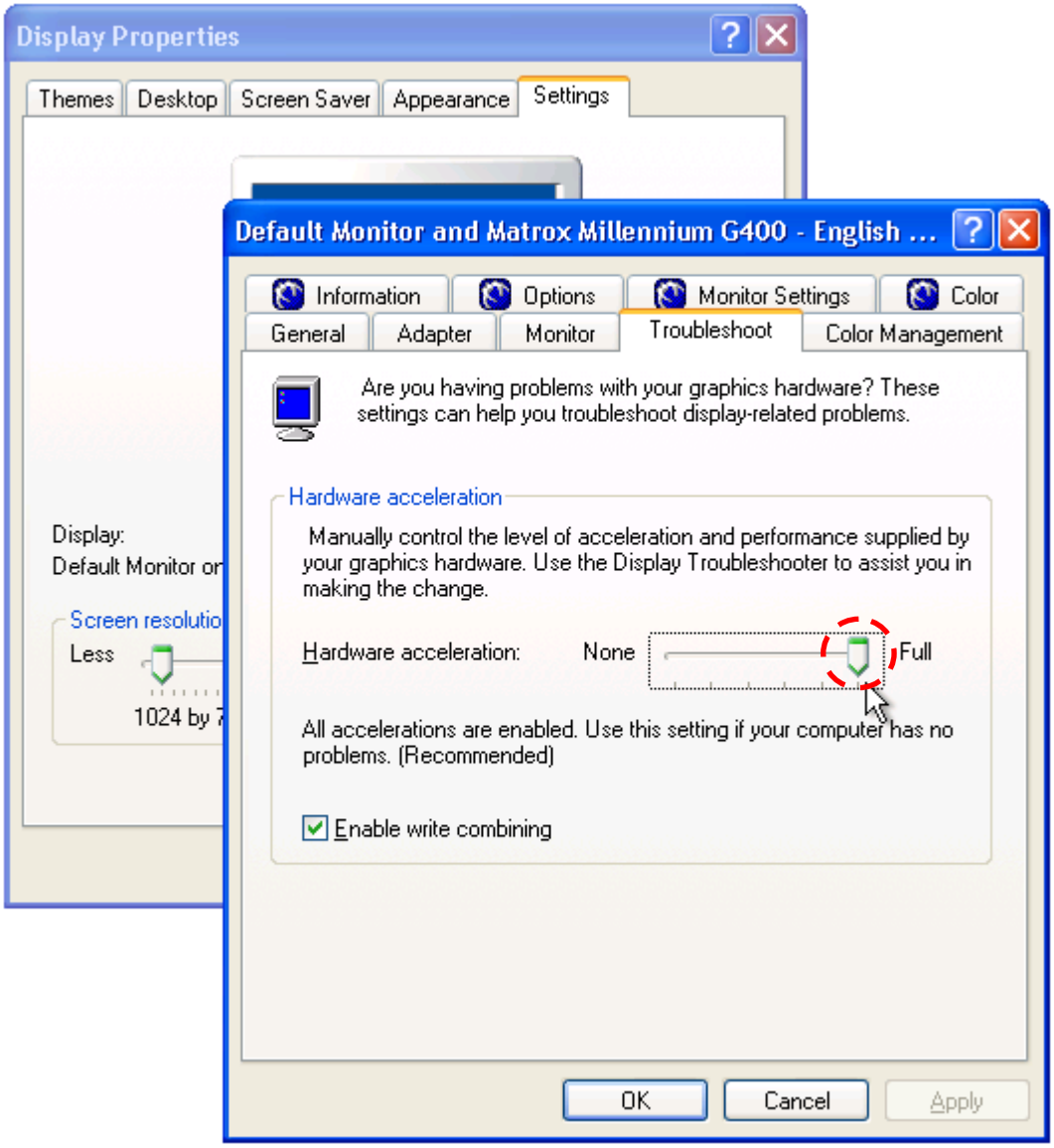


Figure 7-12: Enable Full Hardware Acceleration on WindowsXP guest

Enabling Hardware Cursor Support

Please follow the following steps to enable native hardware cursor support on Windows platforms:

1. Install latest Matrox G400 drivers.
2. Reboot computer.
3. Right click on “My Computer” and select “Properties”.
4. Click on “Advanced’, “Performance”, and then on “Settings…”.
5. Uncheck “Show shadows under mouse pointer” checkbox.
6. Click on “Apply”

7.6 Super IO Devices: Winbond W83627HF SIO / ITE 8712 SIO

Device models of the Super IO device contain the keyboard, PS/2 mouse, floppy, COM1, COM2, LPT1, IR, fan, GPIO, MIDI, and joystick devices, as well as PCI support and control information. The COM1 and COM2 devices create named-pipes "*SimNow.Com1*" and "*SimNow.Com2*" and send all serial communication through these.

Interfaces
The Super IO device model has a single interface connection, and is connected to the LPC connection of the Southbridge device.

Initialization and Reset State
The following conditions represent the keyboard and/or mouse during initialization and reset state:

- A20 and reset released.
- Mouse scaling set to 1.
- Mouse resolution set to 4.
- Stream mode off.
- Mouse sample rate set to 100.
- All sticky keys released.
- Keyboard output port set to 0xDF.

The floppy is initialized with no drive image present. Reset clears the controller to an idle state. If an image is loaded, reset does not unload the image.

COM1 and COM2 are initialized with 9600 Baud, no parity, 8-bit words, 1 stop bit, and interrupts off.

The parallel port initializes with the data and control ports set to zero. Reset clears these ports to their initial values.

The following devices have no functionality behind them at this time, with the exception of their configuration registers. These registers are initialized and reset to the values specified in the Super I/O specification:

- IR
- GPIO
- MIDI
- Joystick
- Fan

Contents of a BSD

- Keyboard and Mouse

- Floppy
- COM1 and COM2
- LPT1
- IR
- GPIO
- MIDI
- Joystick
- Fan

All devices store their current state in the BSD files, as well as any data that may be buffered at the time of the save. Register content is also saved for all devices.

Configuration Options

The Super I/Os have the capability of setting device breakpoints on an event basis. In this case, the event is the sequence of writes to access the Super I/O's device configuration registers. Selecting the *PNP Lock/Unlock Registers* option in Figure 7-13 activates the breakpoint anytime the lock and unlock sequence is hit. The other option is to set breakpoints to trigger whenever any of the device configuration registers are accessed.

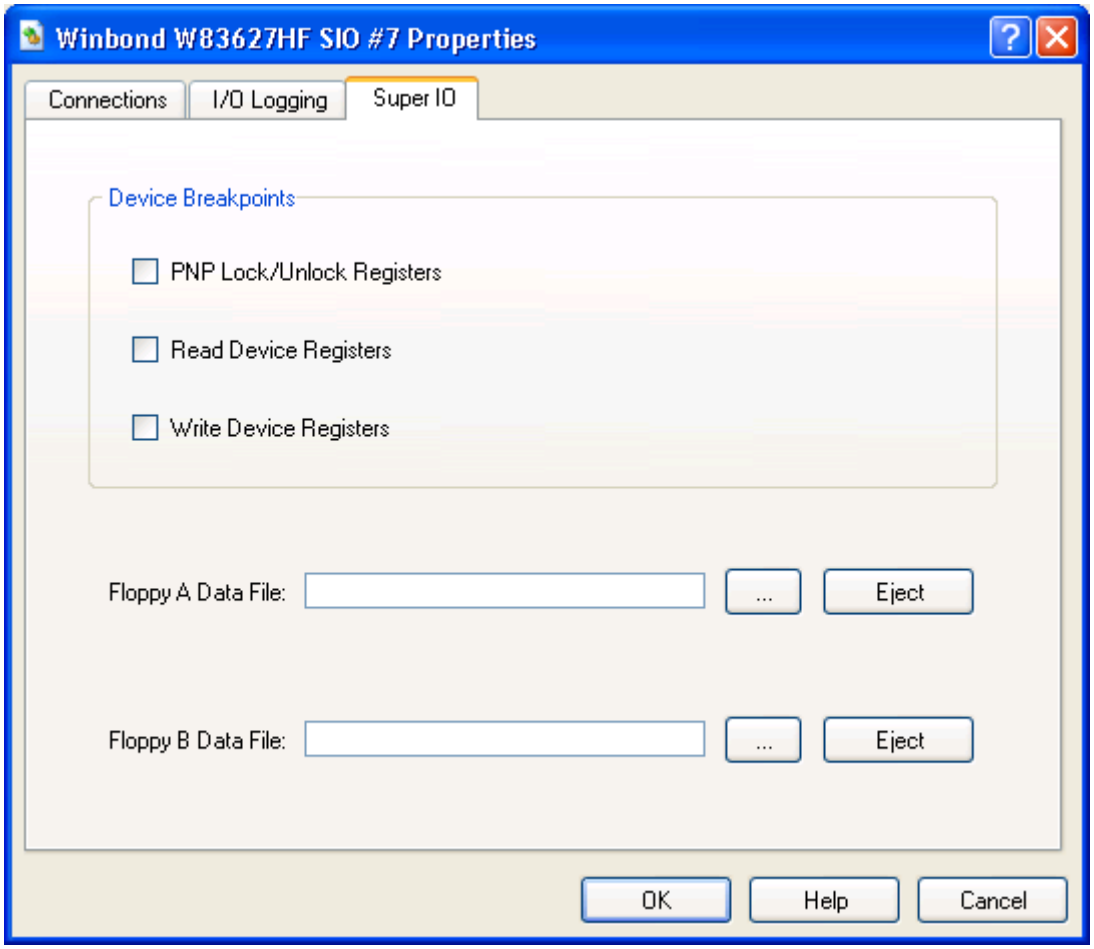


Figure 7-13: Super IO Properties Dialog: Winbond W83627HF

Floppy Configuration Options
The floppy is capable of reading disk images of real floppies created with the *DiskTool* Utility, described in Section 13, on page 161. To use an image, first create an image file with *DiskTool* and then specify the floppy image file in the Super I/O configuration dialog page.

Difference from Real Hardware
Keyboard, Mouse, Floppy, COM1 and COM2 differ from real hardware. Baud rate, parity, and stop bits are ignored. Communication is always available. Baud rate timing is approximate. Modem status and line status always show the device is ready.

The default values of the control registers are read-writable or read only as defined by the appropriate Super IO specification.

7.7 Memory Device

The memory device enables you to add memory devices to the system. You can configure the memory device for emulation of ROM or dynamic memory. You specify the total memory size and the beginning address to which the device should respond.

The memory device can also be configured as a LPC flash device. It currently models 2Mb (SST49LF020A), 4Mb (SST49LF040A), 8Mb (SST49LF080A) and 16Mb (SST49LF160C) flash memory devices. Note that we support two command sequences used generally by flash memory - SST and ATMEL. User should configure the flash memory to the appropriate command sequence to get desired results. The SST49LF160C device uses the ATMEL command sequence while SST49LF020A/SST49LF040A/SST49LF080A use the SST command sequence.

Interfaces
The memory device has a general-purpose interface that you can connect to any other type of port. No selection is necessary when connecting this memory device to another device.

Initialization and Reset State
The default state of the device is a RAM memory device that is at a base address of 0x00000000 and a size of 4 Gigabytes. The memory has no default content. When an initialization file is specified, the memory device's contents contain the data from that binary file.

After a reset, the memory device reverts back to the initialization file contents.

Contents of a BSD
The contents of memory, as well as all configuration information, are stored in the BSD.

Configuration Options
The first field of the *Memory Configuration* tab, shown in Figure 7-14, is the base address of the device in a hexadecimal value.

The second field is the total size of the memory device, given in decimal value for the number of 32-Kbyte blocks you would like created (32-Kbyte blocks are used because non-initialized memory is dynamically allocated when addressed in 32-Kbyte chunks).

The third field is the name of the binary file you use to initialize the memory contents. The device initializes memory for the content length of the file. If you specify a 512-Kbyte ROM and use a 256-Kbyte image file, the first 256 Kbytes are initialized. The Init File selection comes with a browse button for easier selection.

Selecting the *Read-Only* option turns the memory device into a ROM. Writes to the device are ignored when the *Read-Only* option is selected.

Selecting the *System BIOS ROM* option tells the memory device it is the system BIOS. The memory device only responds to memory address ranges accompanied by a chip-select that is generated by the Southbridge device.

Selecting *Flash Mode* option tells the memory device that it is configured as a flash memory device. There are two command sequences supported by our flash memory device - SST and ATMEL, which can be selected by the drop down below.

Selecting the *Memory Address Masking* option indicates that the address received by the memory device is masked by a bit mask with the same number of bits as the size of the memory device (e.g., a 256-Kbyte ROM uses an 18-bit mask, or it is masked by 0x003FFFF). This enables the ROM to be remapped dynamically into different memory address ranges in conjunction with the aforementioned chip-select.

Selecting the *Initialized unwritten memory to (hex):* option initializes otherwise not initialized memory, with a separate field for specifying the byte to use for initialization.

Selecting the *Memory is non-cacheable* option tells the system if the memory described by the device is non-cacheable.

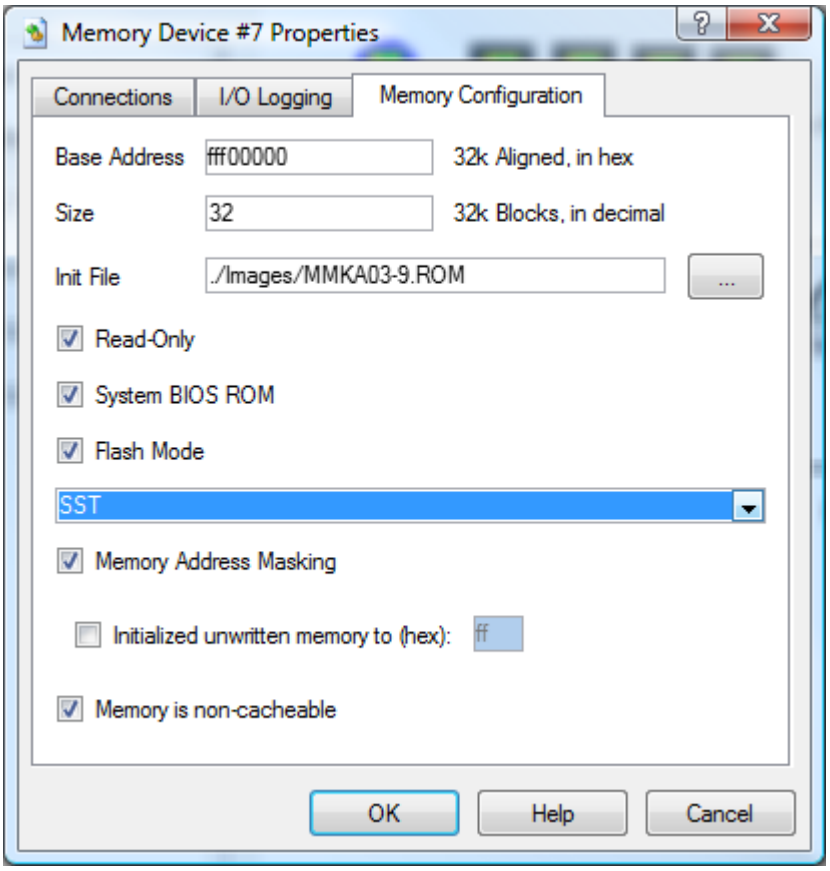


Figure 7-14: Memory Configuration Properties Dialog

Difference from Real Hardware

The memory device differs in that it is a generic memory model. When configured as a BIOS ROM, it does not contain flash-specific information that a modern flash ROM contains (for programming information purposes).

7.8 PCA9548 SMB Device

The PCA9548 is an 8-channel System Management Bus (SMB) switch.

Interface

The PCA9548 has one input port and eight output ports, as well as a programmable interface that directs the switch which output port to forward messages to.

Initialization and Reset State

The PCA9548 has the input value specified in its configuration dialog window.

Contents of a BSD

The PCA9548 saves its *SMB base address* and input pin value.

Configuration Options

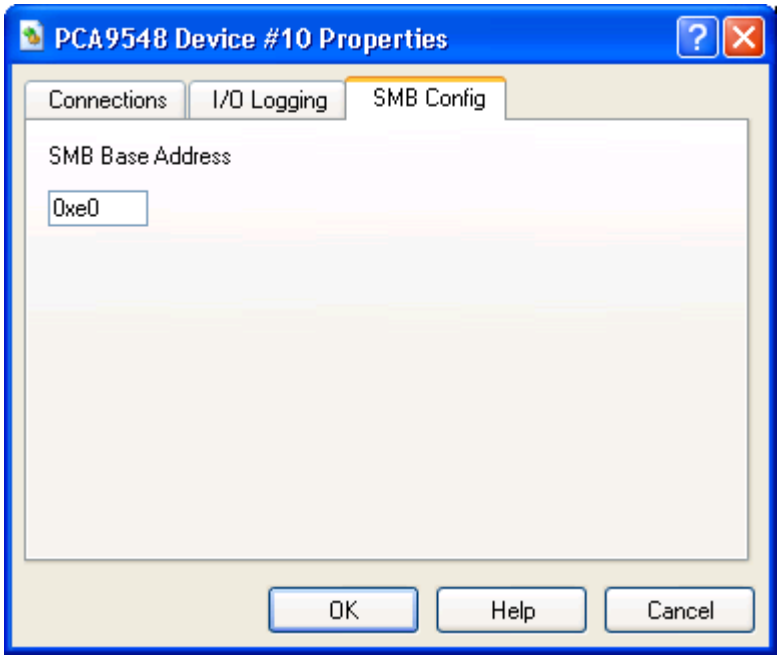


Figure 7-15: PCA9548 SMB Configuration Properties Dialog

The PCA9548 allows you to set its *SMB base address*.

7.9 PCA9556 SMB Device

The PCA9556 is a registered System Management Bus (SMB) interface. When queried from its *SMB base address*, it returns the value of its input pins.

Interfaces
The PCA9556 has one output port.

Initialization and Reset State
The PCA9556 has the input value specified in its configuration dialog window.

Contents of a BSD
The PCA9556 saves its *SMB base address* and input pin value.

Configuration Options

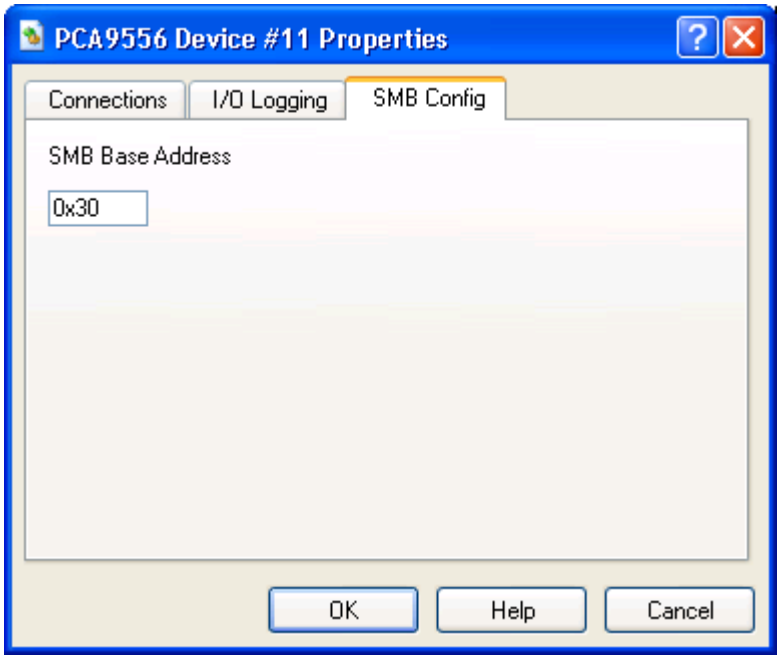


Figure 7-16: PCA9556 SMB Configuration Properties Dialog

The PCA9556 allows you to set its *SMB base address* and input pin values.

7.10 AMD 8th Generation Integrated Northbridge Device

The AMD 8th Generation Integrated Northbridge device supports the AMD 8th generation family of processors - AMD Athlon™ 64 and AMD Opteron™ processors. Although the physical processor chip has a Northbridge built in, for simulation purposes, the Northbridge is considered as a separate unit. Features include HyperTransport™ technology (for coherent and non-coherent connections) and a memory controller. The integrated debugging functions of the 8th generation processors are not included.

Interface

The Northbridge device has several connection points. It has multiple HyperTransport bus ports that connects to the other AMD 8th Generation Integrated Northbridge devices, or to HyperTransport link-capable devices (e.g., AMD-8131 PCI-X device). These ports are mutually exclusive, and should be connected to only one other device. The Northbridge also has a memory bus to the DIMM devices. The CPU bus gives connection points for the CPU. The final port is a system-message bus port for connection with a Log device. A 940-pin 8th generation processor part (AMD Opteron) has three HyperTransport ports; a 754-pin 8th generation processor part (AMD Athlon 64) has one HyperTransport port.

Initialization and Reset State

When first initialized, the Northbridge device is in the default state. This is described in detail in the 8th generation processor PCI register specification.

When reset, the Northbridge device takes on all default register values.

Contents of a BSD

The BSD file contains the contents of all Northbridge registers. It also saves the contents of any tables and the states of all internal devices (the memory controller, HyperTransport table contents, etc.). When the BSD file is read in, all tables are filled with past data, and all states are restored to their saved states.

Configuration Options

Figure 7-17 and Figure 7-18 show configuration options for the Northbridge.

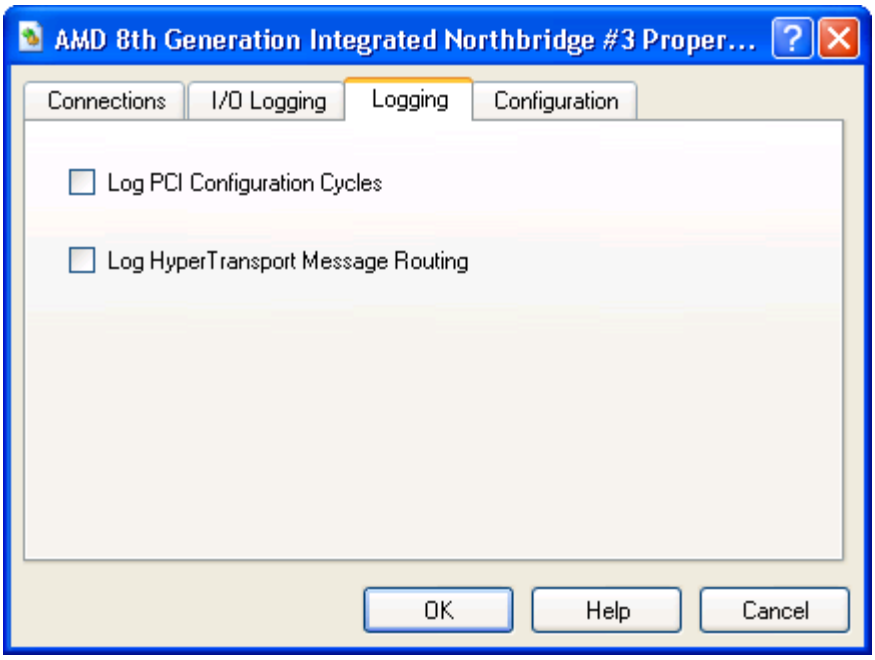


Figure 7-17: Northbridge Logging Capabilities Properties Dialog

If *Log PCI Configuration Cycles* is selected, the device will produce log messages whenever PCI configuration registers are accessed.

If *Log HyperTransport Message Routing* is selected, the device will log HyperTransport messages.

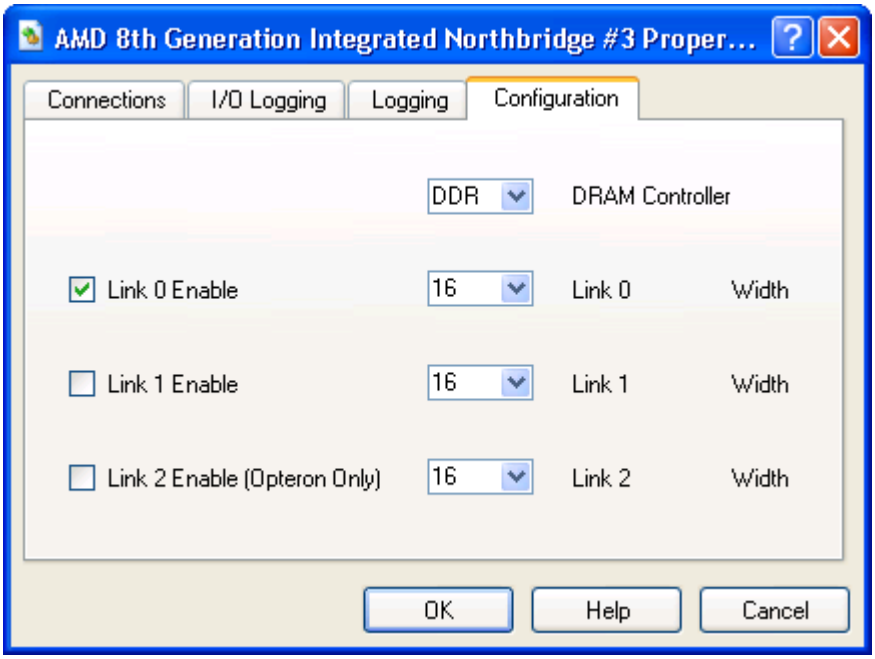


Figure 7-18: Northbridge HT Link Configuration Properties Dialog

If the *DDR DRAM Controller* is selected, the device will support DDR DRAM. In order to use DDR2 DRAM select the *DDR2 DRAM Controller*.

Each HyperTransport link can be enabled separately. Each link can be 8- or 16-bits wide. Only the 940-pin AMD Opteron processor can have three links; a 754-pin AMD Athlon 64 has one HyperTransport port.

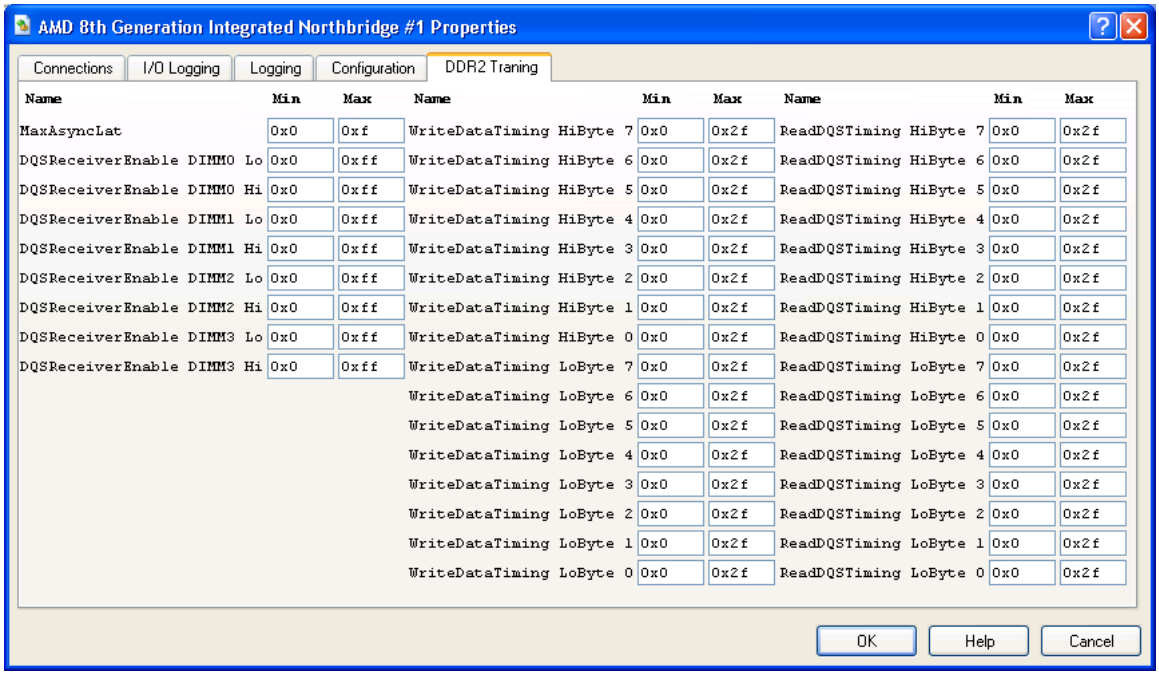

AMD 8th Generation Integrated Northbridge #1 Properties
Connections
I/O Logging
Logging
Configuration
DDR2 Traning
Name Min Max Name Min Max Name Min Max
MaxAsyncLat 0x0 0xf WriteDataTiming HiByte 7 0x0 0x2f ReadDQSTiming HiByte 7 0x0 0x2f
DQSReceiverEnable DIMM0 Lo 0x0 0xff WriteDataTiming HiByte 6 0x0 0x2f ReadDQSTiming HiByte 6 0x0 0x2f
DQSReceiverEnable DIMM0 Hi 0x0 0xff WriteDataTiming HiByte 5 0x0 0x2f ReadDQSTiming HiByte 5 0x0 0x2f
DQSReceiverEnable DIMM1 Lo 0x0 0xff WriteDataTiming HiByte 4 0x0 0x2f ReadDQSTiming HiByte 4 0x0 0x2f
DQSReceiverEnable DIMM1 Hi 0x0 0xff WriteDataTiming HiByte 3 0x0 0x2f ReadDQSTiming HiByte 3 0x0 0x2f
DQSReceiverEnable DIMM2 Lo 0x0 0xff WriteDataTiming HiByte 2 0x0 0x2f ReadDQSTiming HiByte 2 0x0 0x2f
DQSReceiverEnable DIMM2 Hi 0x0 0xff WriteDataTiming HiByte 1 0x0 0x2f ReadDQSTiming HiByte 1 0x0 0x2f
DQSReceiverEnable DIMM3 Lo 0x0 0xff WriteDataTiming HiByte 0 0x0 0x2f ReadDQSTiming HiByte 0 0x0 0x2f
DQSReceiverEnable DIMM3 Hi 0x0 0xff WriteDataTiming LoByte 7 0x0 0x2f ReadDQSTiming LoByte 7 0x0 0x2f
WriteDataTiming LoByte 6 0x0 0x2f ReadDQSTiming LoByte 6 0x0 0x2f
WriteDataTiming LoByte 5 0x0 0x2f ReadDQSTiming LoByte 5 0x0 0x2f
WriteDataTiming LoByte 4 0x0 0x2f ReadDQSTiming LoByte 4 0x0 0x2f
WriteDataTiming LoByte 3 0x0 0x2f ReadDQSTiming LoByte 3 0x0 0x2f
WriteDataTiming LoByte 2 0x0 0x2f ReadDQSTiming LoByte 2 0x0 0x2f
WriteDataTiming LoByte 1 0x0 0x2f ReadDQSTiming LoByte 1 0x0 0x2f
WriteDataTiming LoByte 0 0x0 0x2f ReadDQSTiming LoByte 0 0x0 0x2f
OK
Help
Cancel

Figure 7-19: Northbridge DDR2 Training Properties Dialog

When the *DDR2 DRAM Controller* is selected and DDR2 DRAM is being used you can manually modify these values to verify the correctness of the DDR2 training algorithmn.

The *DDR2 Training Properties Dialog* contains the lowest and highest values that the BIOS can program into these registers. While these registers are programmed out of bounds DRAM access will be corrupted.

Note the *DDR2 Training Properties Dialog* is only useful for BIOS developer and the values should only be modified and used by BIOS developers.

Log Messages

If *Log PCI Configuration Cycles* is selected, the device produces log messages whenever the PCI configuration data register (0xCFC) is accessed. Log files can get **very** large. Reads from this I/O-mapped register produce PCI CONFIG READ messages, and writes to the register produce PCI CONFIG WRITE messages. The formats of the PCI CONFIG READ and PCI CONFIG WRITE messages are as follows:

```
PCI CONFIG READ Bus a, Device b, Function c, Register d, Data e
PCI CONFIG WRITE Bus a, Device b, Function c, Register d, Data e
```

where a, b, c, d, and e are all hexadecimal numbers.

The data value, e, is always one byte (two hex digits) in width. The device will log multiple messages for PCI configuration accesses that are greater than one byte in width. For example, a dword read of 0x11223344 from PCI configuration register 0x40 of device 7, function 1 on bus 0 would produce the following log messages:

```
PCI CONFIG READ Bus 0, Device 7, Function 1, Register 40, Data 44
PCI CONFIG READ Bus 0, Device 7, Function 1, Register 41, Data 33
PCI CONFIG READ Bus 0, Device 7, Function 1, Register 42, Data 22
PCI CONFIG READ Bus 0, Device 7, Function 1, Register 43, Data 11
```

Differences from Real Hardware

The Northbridge device differs from the real hardware in that the simulator does not support the debug hardware registers. The device also does not support memory-interleaving by node, though this will change in the near future. The device will differ in those things that are of a timing-related nature, such as setting of bus speeds. Full probe transactions are not modeled. Registers that deal with items outside of the testing of transfer protocols at the register level are not functional (buffer count registers, etc.). They are present and read/write able, but do not effect the simulation.

7.11 AMD-8111™ Southbridge Devices – IO Hubs

The Southbridge devices provide the basic I/O Southbridge functionality of the system. Features include a PIO-mode IDE controller, register set for the USB controller(s), an LPC/ISA bridge, a system-management bus controller, IOAPIC bus bridge if applicable, and legacy AT devices (PIC, PIT, CMOS, timer, and DMA controller). The legacies AT devices have the standard behavior and IO addresses unless otherwise noted.

Interfaces

The Southbridge devices have several connection points. Possible connection points include a PCI bus, a SMB bus, a LPC bus, an INT/IOAPIC bus for interrupt signaling, and ISA and HyperTransport ports depending on the device type. The PCI bus acts as a host bus (AMD-8111). The SMB connects to devices such as the DIMM or the SMB hub. The LPC bus provides connectivity to devices such as Super IO's and BIOS ROMs. A HyperTransport port is used for main connectivity for the AMD-8111 device to the reset of the system.

Initialization and Reset State

When first initialized, the Southbridge devices are in the default state. This is described in detail in the respective datasheets. The legacy CMOS sub device initializes to all zeroes.

When reset, a Southbridge device takes on all default register values as above. The exception to this is that the CMOS contents remain the same.

Contents of a BSD

The BSD file contains the contents of all registers. It also saves the contents of any buffers, and states of all internal devices (HDD controllers, PIT, PIC, etc.). When the BSD file is read in, all buffers are filled with past data, and all states are restored to their saved states.

Common Configuration Options

The USB dialogue window, shown in Figure 7-20, gives the user the ability to enable or disable USB ports of the USB controller. USB devices which are connected to disabled USB ports won't be identified and detected by an operating system.

For instance, in Figure 7-20 the USB Port 0 is disabled and USB Port 1 and 2 are enabled.

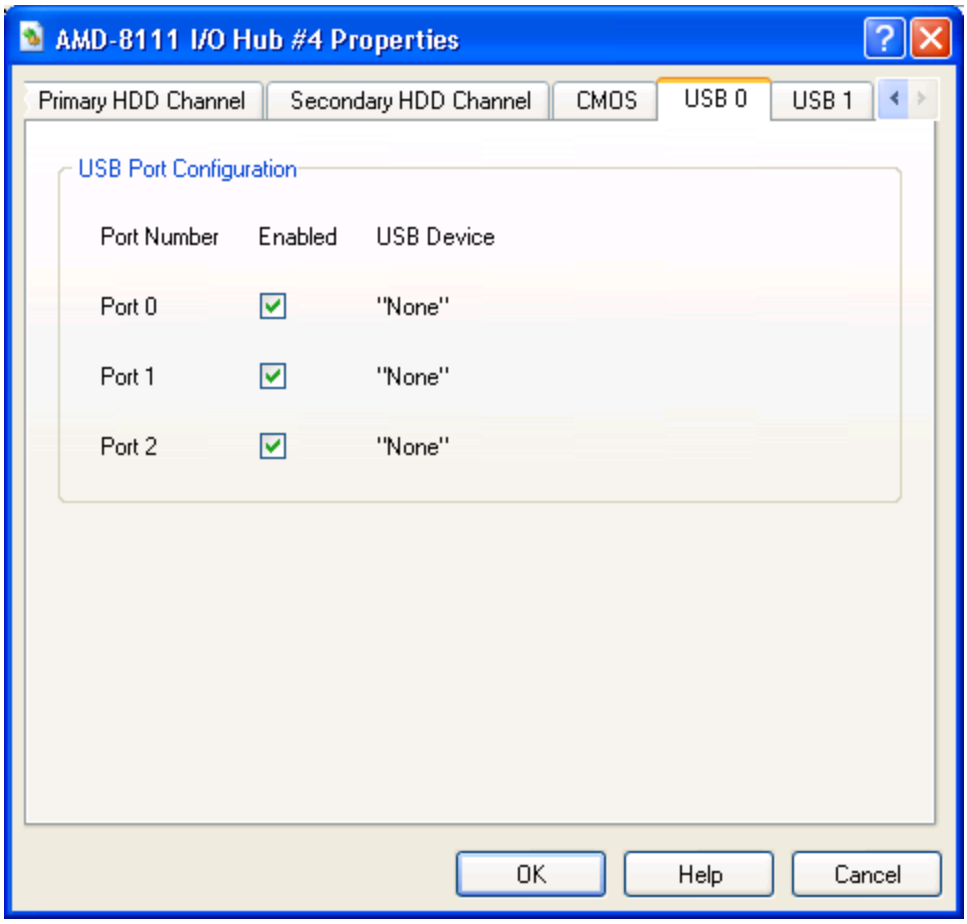


Figure 7-20: USB Properties Dialog (AMD-8111™ Southbridge)

The CMOS dialogue window, shown in Figure 7-21, gives the user the ability to change the contents of CMOS. When first created, the CMOS contains all zeroes to force a CMOS checksum error, resulting in the default settings being loaded by BIOS. The alternative to this is loading a binary file containing the CMOS desired data. The user can create this file by entering changes and using the save feature to create the binary file.

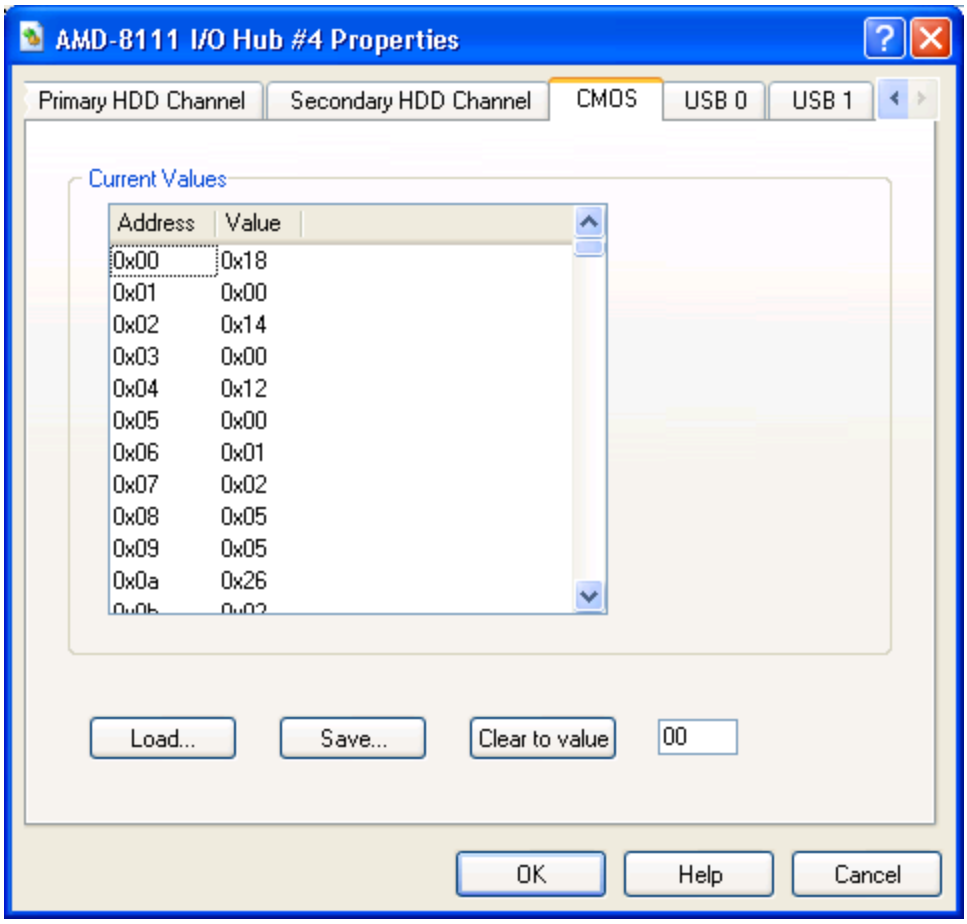


Figure 7-21: CMOS Properties Dialog (AMD-8111™ Southbridge)

The *Primary HDD Channel* and *Secondary HDD Channel* tabs, shown in Figure 7-22, contain the same information for each hard drive channel. The user has two options for drive simulation: an image of a hard drive created with DiskTool (see Section 13 on page 161), or use of a real hard disk. Using a real drive requires Windows® 2000 and a drive that is able to be isolated (locked) from the rest of the system. You cannot use the drive(s) that the OS and/or the simulator reside on. To use a drive image, enter a file name in the *Image Filename* field. A browse window is activated by pressing the right-most button.

All disk devices (Primary Master, etc.) by default have the disk journaling feature turned on, which allows simulations to write to the disk image during normal operation and not affect the contents of the real disk image. This is useful for being able to kill a simulation in the middle, for multiple copies of the simulator running at the same time, etc. Journal contents are saved in BSD checkpoint files but **lost if you don't save** a checkpoint before exiting. To change journal settings or commit journal contents to the hard disk image, go to the *Device View Window*, then the AMD-8111 Southbridge, then the configuration for the hard disk in question on either the Primary or Secondary IDE controller. Here you can either commit the contents of the journal to the hard-disk image or turn off journaling for the hard disk image in question.

Turning off journaling is recommended during the installation process for an operating system.

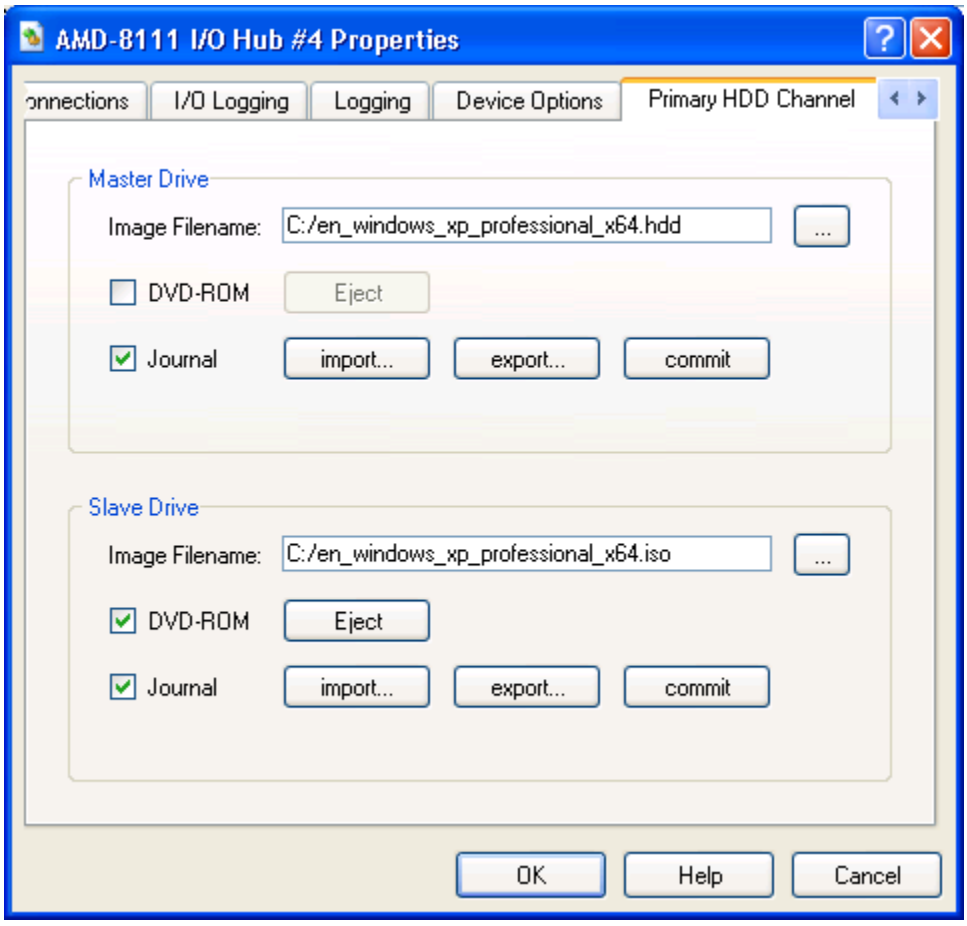


Figure 7-22: HDD Primary Channel Properties Dialog (AMD-8111 Southbridge)

Device Options

The AMD-8111 device has specific configuration requirements that relate to device option type and HyperTransport information.

The *Default Base Unit ID* is a way of telling the device of the strapping option for ID selection.

The *Generate HT Messages for Interrupts* selection specifies whether interrupts go out the HyperTransport port in a HyperTransport format, or out the INT/IOAPIC bus as a classic interrupt pin.

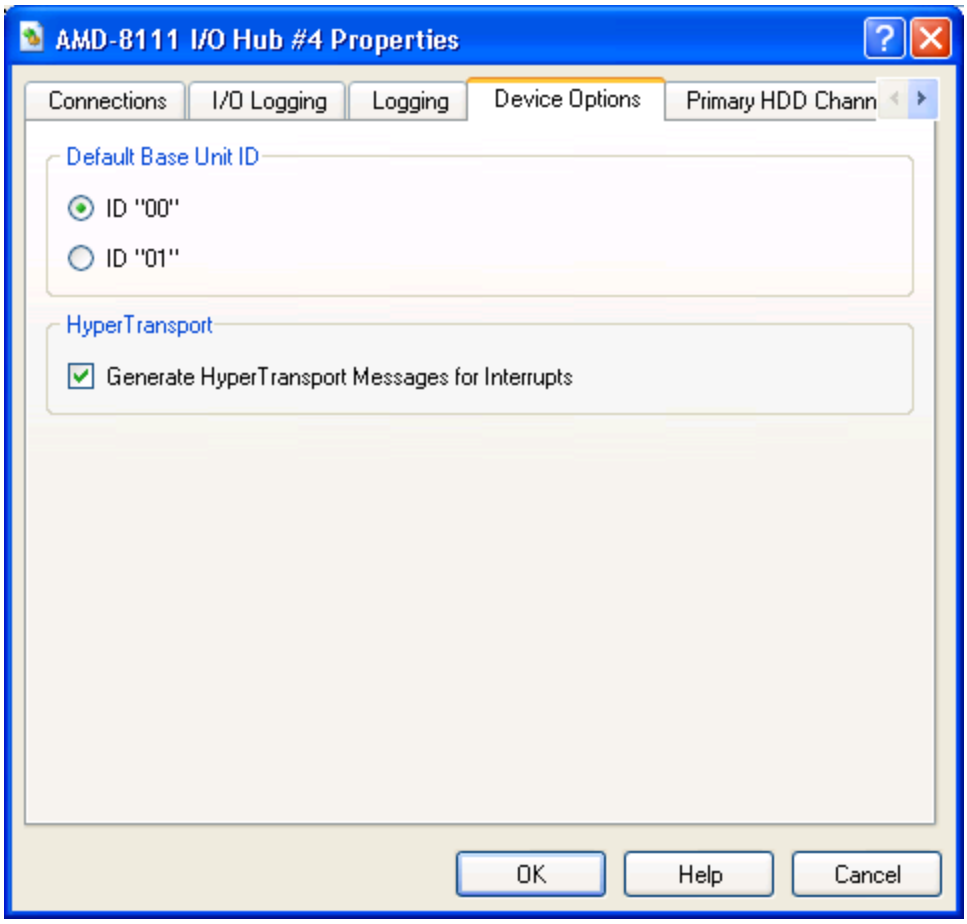


Figure 7-23: Device Options Properties Dialog (AMD-8111 chipset)

Log Messages

The AMD-8111 device produces log messages to the *Message Log Window* as specified by the options in the *Logging Option* tab, shown in Figure 7-24. The device can log I/O-mapped Transactions, Memory-mapped Transactions, and SMI and SCI assertions.

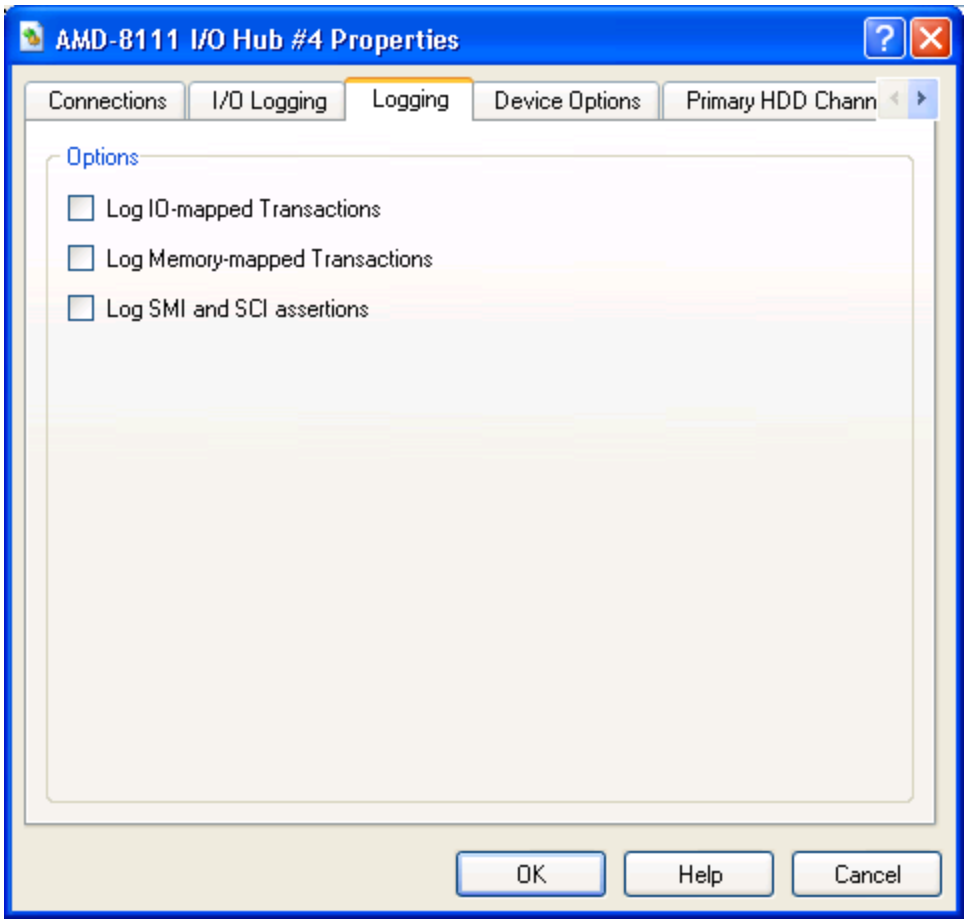


Figure 7-24: Logging Options Properties Dialog (AMD-8111 chipset)

Differences from Real Hardware

The AMD-8111 Southbridge device differs from other devices mainly in those items that deal with real-time operation. Those items cannot be modeled in the current simulator. The model does not include any of the power-management registers. The functionality of the USB 2.0 controller is also absent (PCI registers and memory-mapped registers are the only portion present).

For experimental purposes, the AMD-8111 Southbridge device supports an optional IOMMU (based on IOMMU spec. 1.2) that can be enabled and disabled via the automation command "*8111.SetIOMMU 0|1*". The addition of this block to the device model does not reflect any real or planned hardware. When enabled, the AMD-8111 device's IOMMU PCI registers live in a capability block of the PCI Bridge. When enabled, the AMD-8111 device's IOMMU delivers interrupts via PCIINTD. The AMD-8111 device doesn't support PCI Express. This limits the number of distinct requester ID's available (Three requester ID's: legacy LPC, legacy PCI, internal IDE controller). There are no SimNow PCI models that implement MSI. This means the only APIC-style interrupts the IOMMU can intercept are from a single requester ID, the AMD-8111 device's internal IOAPIC.

7.12 PCI BUS Device

The PCI Bus device enables you to add PCI devices to the system. You can configure the PCI Bus device to provide any number of PCI slots for one to six connections. You configure each PCI slot on the PCI Bus by setting its device number and base IRQ-routing pin.

Interfaces
The PCI Bus device has two types of interfaces, a bus interface and one or more slot interfaces. The bus interface connects to a device that provides a PCI bus, such as a Northbridge. Each PCI-slot interface is capable of connecting to a PCI device, such as a PCI video controller.

The PCI bus behaves somewhat differently than other AMD SimNow devices. First, the connection points are not all centered in the middle of the icon; instead each connection point has a discrete location around the perimeter of the icon to provide a visual indication that each PCI device is connected to a different PCI slot. Second, the connection points are exclusive; that is, only one device can connect to each connection point on the PCI bus, because in a real system one cannot install two PCI cards into a single PCI slot. It is planned that these new behaviors will be used in other devices when required.

Initialization and Reset State
The default state of the device has all slots disabled. This is because each platform configures its PCI Buses in specific ways that make it impossible to provide a generic default.

Since the PCI Bus device does not include any state that is altered during the course of a simulation, after a reset, the PCI Bus device remains unchanged

Contents of a BSD
The configuration of the PCI bus, including which slots are enabled, the device ID for each slot and the base IRQ-routing pin for each slot, and the connection points, are saved in the BSD.

Configuration Options
Figure 7-25 shows the *PCI-Bus configuration* options.

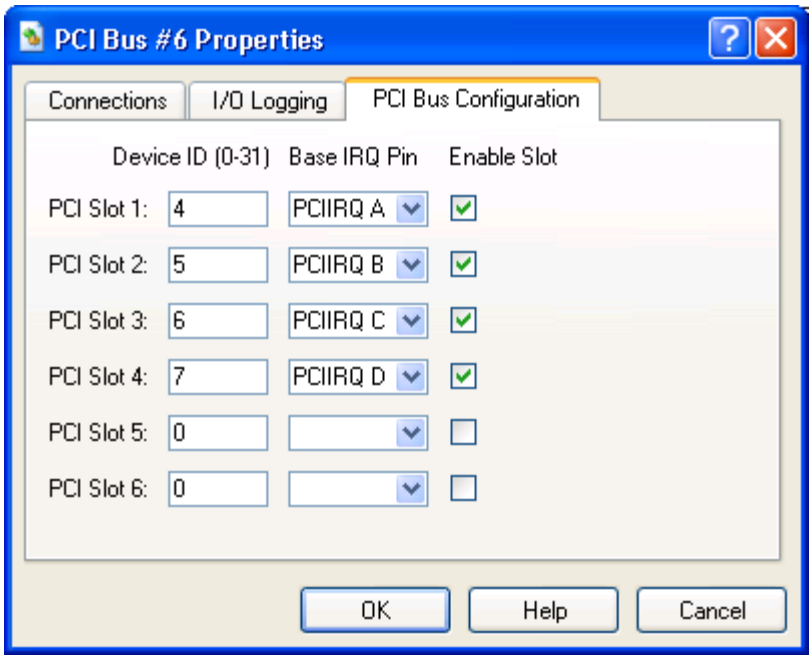


Figure 7-25: PCI Bus Properties Dialog

The first field is the *Device ID* of the slot. This value may range from zero to 31. The second field is the *Base IRQ Pin* for the slot. You can choose from pin A, B, C, or D.

The third field is an *Enable Slot*. By default, all slots are disabled. One cannot disable a slot that has a device connected to it.

Differences from Real Hardware

The PCI Bus device differs from other devices in that it is a generic model. We do not simulate PCI down to a clock-accurate level, so devices do not arbitrate for bus ownership or insert wait states, for example.

7.13 AMD-8131™ PCI-X® Controller

The AMD-8131 PCI-X Controller is a HyperTransport tunnel that provides two PCI-X buses and two IOAPICs. These PCI-X buses may or may not be configured as hot-plug-capable, depending on the platform.

Interfaces

The AMD-8131 has two types of interfaces, HyperTransport and PCI buses. It has two HyperTransport links, HT0 and HT1, that can connected to other non-coherent HyperTransport link-capable devices. The PCI bus interfaces in the AMD-8131 must be connected to a PCI bus device, which provides the Slot interfaces with which to connect devices for simulation.

Initialization and Reset State

When first initialized, the AMD-8131 tunnel is in its default state. This is described in detail in the AMD-8131 datasheets. Each bridge defaults with hot-plug functionality disabled.

When reset, the AMD-8131 takes on all default register values.

Contents of a BSD

The entire configuration of the AMD-8131 device, including all state and registers for its sub devices, is saved in the BSD.

Configuration Options

The only configuration options for AMD-8131 are to enable or disable hot-plug for each of its PCI-X bridges, as shown in Figure 7-26. You cannot enable or disable hot-plug after a simulation has already begun.

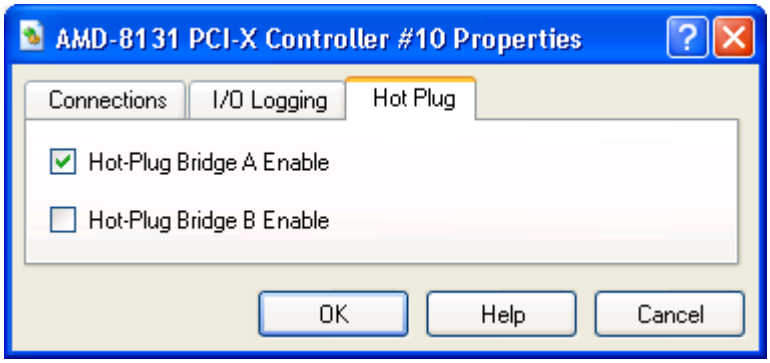


Figure 7-26: AMD-8131™ Device Hot Plug Configuration

Differences from Real Hardware

Clock-sensitive functionality, like setting bus speeds, is not supported. Neither are system errors or power management.

7.14 AMD-8132™ PCI-X® Controller

The AMD-8132 PCI-X Controller is a HyperTransport tunnel that provides two PCI-X buses and two IOAPICs. These PCI-X buses may or may not be configured as hot-plug-capable, depending on the platform.

Interface

AMD-8132 has two types of interfaces, HyperTransport and PCI buses. It has two HyperTransport links, HT0 and HT1 that can connect to other HyperTransport link-capable devices. Either HyperTransport link can be set to be the upstream HyperTransport link. The PCI bus interfaces in the AMD-8132 must be connected to a PCI Bus device, which provides the Slot interfaces with which to connect devices for simulation.

Initialization and Reset State

When first initialized, AMD-8132 device is in its default state. This is described in detail in the AMD-8132 datasheet. Each bridge defaults with hot-plug functionality disabled.

When reset, AMD-8132 takes on all default register values.

Contents of a BSD

The entire configuration of the AMD-8132 chipset, including all state and registers for its sub devices, is saved in the BSD.

Configuration Options

The *Hot Plug* tab options for AMD-8132 are to enable or disable hot-plug for each of its PCI-X bridges, as shown in Figure 7-27. You cannot enable or disable hot-plug after a simulation has already begun.

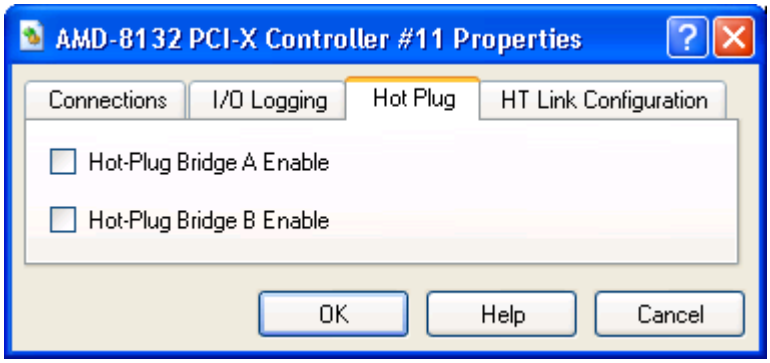


Figure 7-27: AMD-8132™ Device Hot Plug Configuration

Figure 7-28 shows the *HT Link Configuration* options.

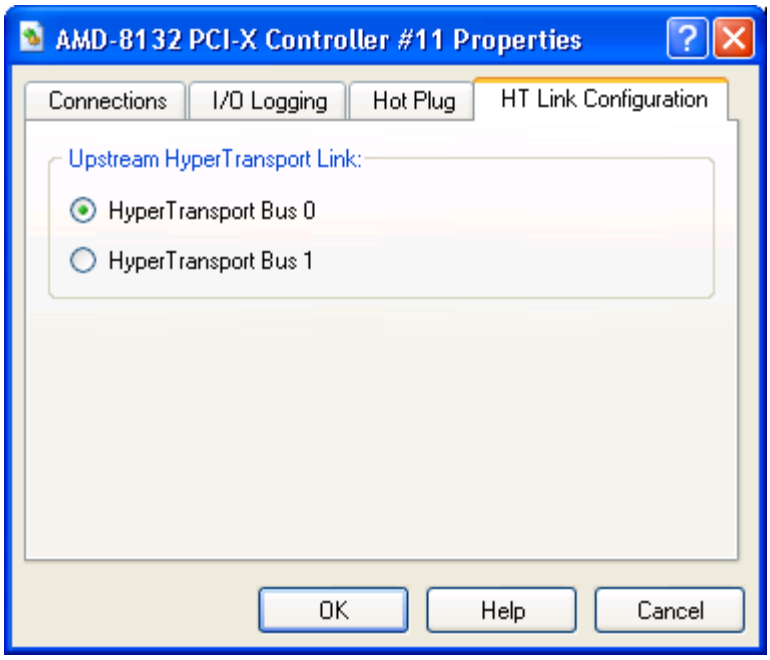


Figure 7-28: AMD-8132 Properties Dialog

The *Upstream HyperTransport Link* selection, shown in Figure 7-28, specifies the *HyperTransport Bus* that will be used as a upstream link.

Differences from Real Hardware

Clock-sensitive functionality, like setting bus speed, is not supported. Neither are system errors, nor power management.

7.15 PCI-X Test Device

This PCI-X Test Device model provides a simulation of a generic PCI-X device. Its main purpose is to provide BIOS programmers with a tool to test the PCI-X configuration cycle. This device is implemented as a single-function device.

Interface
The interface varies from system to system. In the AMD Athlon 64 or AMD Opteron processor-based system configurations, it can be connected to AMD-8131 PCI-X or AMD-8111 Southbridge devices.

Initialization and Reset State
At creation and reset states, the PCI-X device registers have the default hard-coded values. By default, the PCI-X device is set to have no I/O, memory-space and interrupt capability. The PCI-X device has a default Device ID and Vendor ID. At reset, the device configuration does not change and the values from the device configuration will be eventually read into the PCI-X registers when the configured system is restarted.

Contents of a BSD
PCI-X register and interrupt signals are saved in the BSD.

Differences from Real Hardware
This is a generic PCI-X device. It doesn't have real a memory buffer and I/O buffer. For memory and I/O space transaction, if the transaction belongs to this device's memory or I/O address range, the PCI-X device simply outputs a message to the *Log Window* which identifies its memory or I/O cycle.

Interrupt can be de-asserted by doing an I/O transaction. Interrupts can also be de-asserted manually by using the debugger.

7.16 AMD-8151™ AGP Bridge Device

The AMD-8151 AGP Bridge Device tunnel is a HyperTransport tunnel that provides an AGP bridge. In general, AMD-8151 would be connected in a non-coherent HyperTransport chain between the host bridge and the Southbridge.

Interface

The AMD-8151 has three types of interfaces, HyperTransport, AGP, and INT/IOAPIC buses. The AMD-8151 has two HyperTransport links, HT0 and HT1, that can connect to other non-coherent HyperTransport link-capable devices. HT0 should be connected to the upstream link (the one closest to the host bridge) and HyperTransport1 should be connected to the downstream link. The AGP interface should be connected to an AGP graphics device. The INT_IOAPIC bus should be connected to the Southbridge; it routes interrupt signals from the AGP bus to the Southbridge.

Initialization and Reset State

When first initialized or reset, the AMD-8151 registers are set to their default state. This is described in detail in the AMD-8151 datasheet.

Contents of a BSD

The current state of all PCI configuration registers and any internal state variables are saved in the BSD.

Configuration Options

The AMD-8151 device allows you to set its *Revision number* as shown in Figure 7-29.

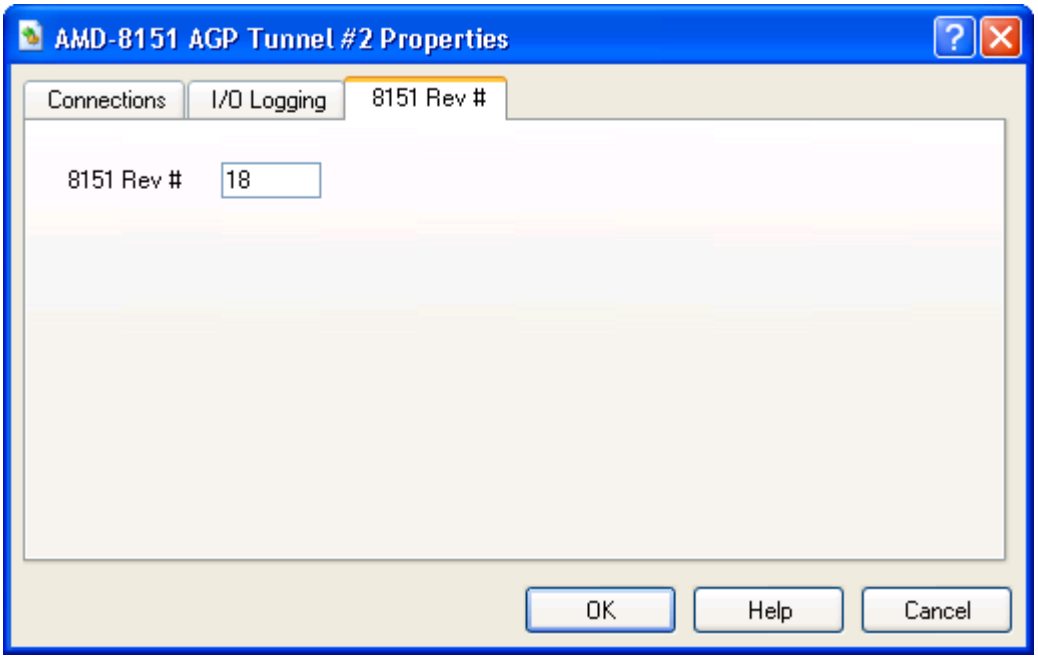


Figure 7-29: AMD-8151™ Device Properties Dialog

Differences from Real Hardware

Clock-sensitive functionality, like setting bus speeds, is not supported. The HyperTransport bus protocol is not simulated.

7.17 Raid Device: Compaq SmartArray 5304

The RAID device uses disk images, which are accessed as simulated volumes by the RAID controller. Storage devices like ATA HDD and RAID are implemented with concepts like disk-block cache, journaling, file, and memory stores. This page describes journaling in more detail.

A simulated volume in the RAID device is represented by an image file and one or more optional journals. The combination of an image and zero or more optional journals is used to hold the contents of a simulated volume. While creating a volume assign a disk-image file to it (e.g., "*raid.image 0 imagefilename").* One or more additional journals can be added to the image file. The image file uses a data block to store the data, and the journal files use sparse indexing to hold just the blocks that have been changed. Not only does journaling provide an efficient way to access the data blocks in the simulated volume, but it also gives the user the flexibility to change the data-block size.

Journals can be created either in-memory or as file, depending on the use of *"addjournal*" command. RAID device supports multi-level journaling; i.e., for a created volume, the user can add multiple journals (however, one cannot add a journal after an in-memory journal). Conceptually, the disk image is equivalent to the image and fixed-journal pair.

Journals grow in size as the volumes associated with them are accessed (writes of data-blocks which haven't been written before). File-based journals are preferred over in-memory Journaling if a large number of writes are going to be made to the simulated volume.

The journal architecture is index-based, consisting of super blocks, index blocks, and data blocks. This provides a hierarchical indexing mechanism, in which data blocks are accessed by their LBA (logical block address).

Several performance mechanisms are implemented in the RAID device, including *Disk Block Cache* and *Last Sector Hit*, which can be viewed at any time using the "*raid.status –v"* command.

AMD tested the RAID device both on SUSE Linux-64 and a 32-bit version of Windows 2003 Enterprise Server, using stock drivers to drive this model. This model emulates devices at the volume level, so that the files used to represent the data correspond to logical volumes, not disks. This model associates one logical volume with one image file. The model does not represent the timing of any real system, because data becomes available almost immediately.

7.18 SMB Hub Device

The SMB hub device is used to connect one SMBus to any of four SMBus branches. The device is programmed via read-byte and write-byte commands on the SMBus where the 7-bit address field is 0x18.

The SMB hub device models the combination of two physical devices manufactured by Philips Semiconductors: the PCA9516 5-channel I^2C hub, and the PCA9556 Octal SMBus and I^2C registered interface. In the simulator's device model the two devices are configurable via *GPIO x enables segment x*, as shown in Figure 7-30.

Interface
The SMB hub has five SMBus interfaces. SMB0 can be connected within the SMB hub to any of the four other SMBuses (SMB[1..3]). Typically, SMB0 is connected to a SMBus connection on a Southbridge device, and the other SMBus ports are connected to other devices in the system.

Initialization and Reset State
When first initialized or reset the SMB hub registers are set to their default state. The internal registers and their default states are described in the PCA9556 data sheet.

Contents of a BSD
The current state of all internal registers and any internal state variables are saved in the BSD.

Configuration Options
The SMB Hub device allows you to enable up to eight GPIO segments (GPIO0 – GPIO7) to connect SMB devices to SMB hub device, as shown in Figure 7-30.

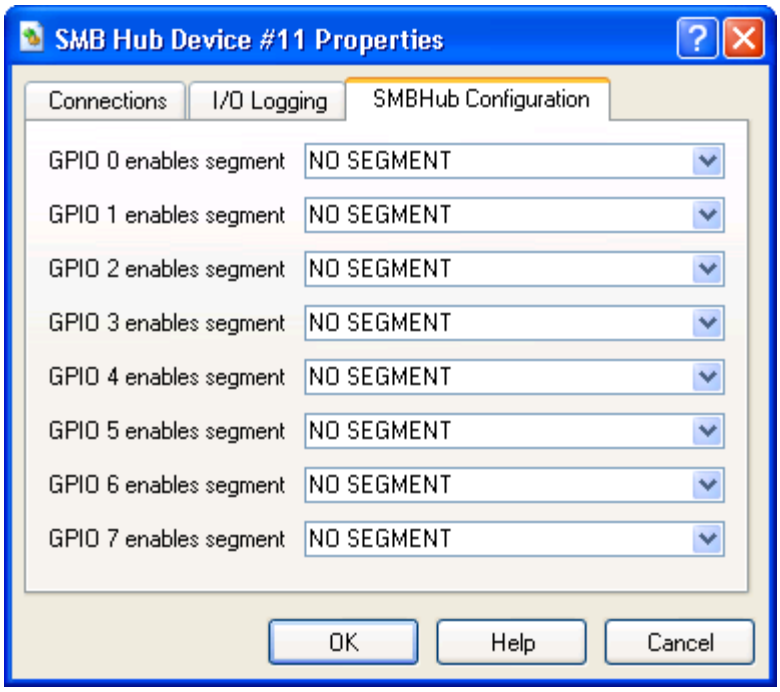


Figure 7-30: SMB Hub Properties Dialog

Differences from Real Hardware

This device model is the combination of two physical devices connected in a specific way. The model attempts to match the functionality of the physical devices from a programmer's perspective. The SMBus protocol is not modeled. Also, the SMBus address of the PCA9556 is programmable based on pin-strapping, whereas this model has a fixed SMBus base address.

7.19 AT24C Device

The AT24C device is a Serial EEPROM device. It can be configured to store 16, 32, or 64Kb of EEPROM. The device has an SMB bus interface for access to its internal registers. It is typically used to store platform specific configuration data.

Interface

The AT24C device has a SMB interface. For example, this device can be connected to a PCA9548 or PCA9556 device (see Section 7.8, "*PCA9548 SMB Device*", on page 84 or Section 7.9, "*PCA9556 SMB Device*", on page 85).

Contents of a BSD

The current state of all internal registers and any internal state variables are saved in the BSD.

Configuration Options

The AT24C device can be configure to store an AT24C16A (16Kb), AT24C32A (32Kb), or AT24C64A (64Kb), 2-Wire Bus serial EEPROM.

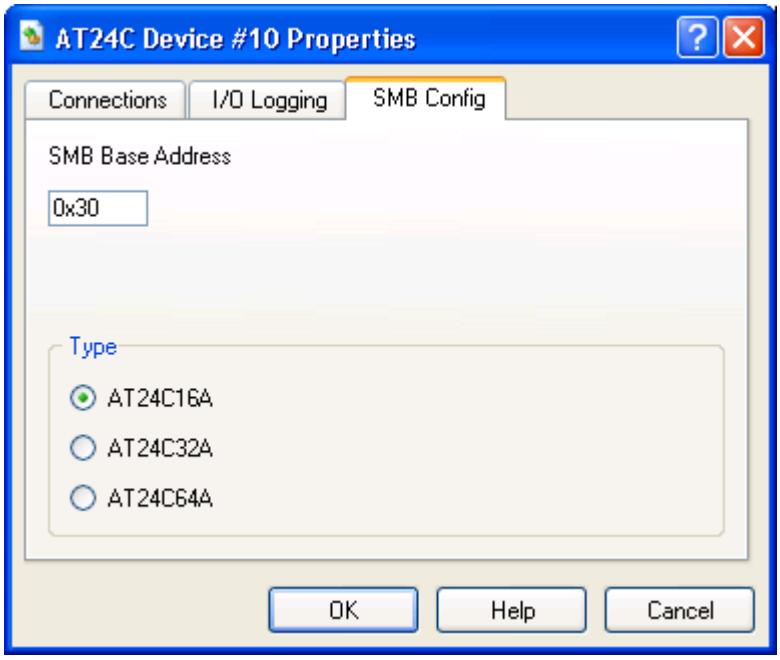


Figure 7-31: AT24C Device Configuration

7.20 EXDI Server Device

This interface, and the instructions contained herein, applies only to the Windows operating system-hosted version of the simulator.

The simulator provides a special device known as the EXDI Server Device. This device can be added to any BSD. When a BSD containing the EXDI Server Device is loaded, the EXtended Debugging Interface becomes available. This allows client debugging software, such as CmdeXdi and the Windows kernel debugger, to interact with the platform being simulated, as if it were a real hardware platform.

The installation of the simulator should provide all the COM registration hooks that are required. If it does not, here are the steps to manually register the EXDI server:

1. Open a command window (run cmd.exe).
2. Change the current directory to the location where the simulator was installed.
3. Execute the command [Regsvr32 exdi64ps.dll]. You should get a message box indicating that registration was successful.
4. Execute the command [Regrgs exdiamdserver.rgs MODULE="path and file name of exdi64ps.dll, usually C:\SimNow\exdi64ps.dll"].

When running the Windows kernel debugger, you must provide command line information that tells the debugger how to attach to the EXDI Server. The command line for this is:

```
kd -kx exdi:clsid={F65E71B3-FEDC-4FA7-A818-5959CD30DD41}
```

7.21 USB Keyboard and USB Mouse Devices

USB legacy emulation is not yet supported by the simulator model. USB 2.0 support is very limited, only basic PCI configuration and memory read and write functionality is available.

By default, the simulator uses the keyboard device model to send user's keystrokes to the simulation. For example, when the user presses Enter with the host mouse on the graphics display window, the simulator sends the internal command, *keyboard.key 0x10 0x80*, to its command interpreter. If the user has a USB keyboard or mouse in his simulation, he can redirect the simulator to use these USB devices for keyboard and mouse input. He does this by modifying the following simulator registry keys: *Gui_Key_Device=usbkey* and *Gui_Mouse_Device=usbmouse* (from the top-level *View→Registry*). With these changes, when the user presses the Enter key in simulation, the simulator will send the internal command, *usbkey.key 0x10 0x80* to its command interpreter. When the user moves the mouse around the simulator display, the simulator will send commands, such as *usbmouse.mousemouve 10 10* to the interpreter.

7.22 XTR Device

XTR is a trace record and playback mechanism that is instrumental for applications that are not dependent on the specific version of the CPU. An XTR trace contains the interaction of the processor with the rest of the system in an XML based log file. The XTR trace file can be played back and could be used to simulate behavior of one or more devices within a system, which in turn may be used to analyze the CPU's performance or to perform conformance analysis between various revs and models of the CPU. XTR may also be used in studies where the behavior of some devices needed but the use of an actual device or its software model is either difficult of impossible due to various constraints.

XTR has two files, a binary file which has the memory dump of the system and an XML based text file which contains the log of the events or messages that go in and out a non-coherent port of the Northbridge, including the DMA signals from devices on the (host's) secondary bus to the DIMM. XTR playback mechanism essentially replaces all the devices including the Northbridge and downwards and feeds the processor with the data present in the XTR XML file. The structure of both binary file and XML file is discussed below.

XTR can be used both in uni-processor (XTR-UP) and multi-processor (XTR-MP) configurations. However, currently only XTR-UP is supported while XTR-MP is under development.

There are two modes of XTR, XTR Record and XTR Playback. The simulator supports both modes and one mode does not necessitate the other. The simulator could be used to record XTR traces only or playback XTR traces generated from other sources as far as the XTR specification is followed correctly (see Section 7.22.4, "*Limitations*", on page 117).

An XTR XML file contains Initialization Data, Events and Instructions. XTR Initialization data stores the state of CPU just before XTR recording is initiated. This data is used to initialize the CPU and memory parameters during Playback (the memory itself is initialized from the contents of the binary file). Any register that does not have corresponding initialization data in XTR XML file will be initialized with zero. XTR events fall into two categories:

- Dormant Events, which record an event occurrence but do not trigger an event during playback.
- Active events that are recorded in XTR file and are actively triggered during playback.

IOR, IOW, MEMR, MEMW, RDMSR are examples of dormant events and INTR, APIC, DMAW, EOT are examples of Active events. XTR Instructions are commands that are injected in the XTR trace to give special instructions during XTR playback. FJMP (Force Jump) is an XTR Instruction.

7.22.1 Using XTR

No special setup for XTR Record is required; XTR can be recorded by using the appropriate automation commands as described in Section A.7.28, "*XTR*", on page 247.

The XTR XML file can easily exceed five Gbytes in size. Please make sure you have enough physical storage before you start XTR Record.

7.22.1.1 Recoding XTR Trace

To record XTR, please enter the following commands in the simulator's console window:

```
1 simnow> xtrsvc.xtrfile <filename.xml>
1 simnow> xtrsvc.xtrenable 1
1 simnow> go [or hit Run on the shell]
```

7.22.1.2 Stop XTR Record

To stop XTR record, please enter the following commands in the simulator's console window:

```
1 simnow> stop [Stop the simulation]
1 simnow> xtrsvc.xtrenable 0
```

7.22.1.3 XTR Playback

For XTR Playback, XTR Northbridge (XTRNB) replaces all the devices including any other Northbridge in the system. Hence for UP-XTR Playback, only AweSim and XTRNB are required. Please refer to Section 7.22.1.3, "*XTR Playback*", on page 111, on how to connect AweSim and the XTRNB device. It is recommended to also include the Debugger device for debugging or logging needs.

To playback XTR, please enter the following commands in the simulator's console window:

```
new
adddevice "Debugger"
adddevice "Awesim Processor"
cpu.type K8
cpu.setname Athlon64
cpu.setnumcores 1
cpu.forcefinegrainedevents 1
cpu.SetStartUpFID 12
adddevice xtrnb
connect "Awesim Processor #0" "CPU Bus 0" "xtrnb #2" "CPU Bus 0"
connect "Awesim Processor #0" "Interrupt / IOAPIC Bus" "xtrnb #2"
"Interrupt / IOAPIC Bus"
cpu.type K8
modifyregistry "System Bus Frequency" "100"
xtrnb.xtrfile <filename.xml>
xtrnb.debug 1
xtrnb.xtrlogfile <filename-playback.log>
SetLogFile <filename.log>
SetLogFileEnabled 1
SetErrorLogFile <filename.errlog>
```

```
SetErrorLogFileEnabled 1
go [or hit Run on the shell]
```

7.22.1.4 Stop XTR Playback

XTR Playback will stop automatically when **E**nd **O**f **T**race (EOT) event is reached. It could also be stopped prematurely by clicking on the *stop* button or by executing the stop automation command.

Initialization and Reset State
XTR Record does not have any special Initialization or Reset state.

Init from BSD
The BSD contents of XTRNB are loaded. The XTR XML file is skipped the number of lines to the last event read and the system prepares itself for playback.

Init from Automation Script
The CPU is initialized from the initialization data in XML and the system prepares itself for playback. This method does not support persistent storage of XTR state to be replayed later.

Reset
The XTR file handle is closed. All the queued events are flushed. Simulated DIMM memory is flushed and unallocated.

Contents of a BSD
XTR Record contains *xtrsvc*, which is described below, in addition to modules in the simulation. For XTR Playback, the BSD is composed of following modules:

```
shell:0     : The shell under which a simulation is executed.
xtrsvc:0    : XTR service which facilitates execution of XTR Playback.
Debug:0     : The SimNow Debugger.
Cpu:0       : AweSim CPU Module. There might be more CPUs for XTR-MP.
xtrnb:0     : XTR Northbridge.
```

In persisted BSD, XTRNB, which is only used during XTR Playback, saves and restores events that have been queued but not triggered yet, DIMM image and internal states of the XTRNB. Complete XTR Playback setup also includes AweSim and optionally the AMD Debugger. Please refer to the documentation of AweSim and AMD Debugger for their respective contents in the BSD file.

XTR Record does not store any contents in the persistent BSD file.

Log Messages
Messages are logged only by XTRNB, which is only used during XTR Playback. Some of the following may only be logged when *xtrnb.debug* is set to enable. Some of the Log messages are:

XTRNB: Attempting to allocate large buffer of size 1074503680

Logged during XTR initialization phase just before XTR tries to allocate memory to simulate DIMM.

XTRNB: Sending APIC initialization data to CPU0

Logged during XTR initialization phase just before APIC memory is initialized.

XTRNB: Write to TSC ignored. Please use M00000010 for writes to TSC

Logged during XTR initialization phase.

XTRNB: CPU0 rejected Initialization SREG XXXXXXXXXX with zeros

Logged during XTR initialization phase and displayed if the initialization data is invalid for the SREG. This may or may not be an error in the initialization data.

XTRNB: CPU0 rejected Initialization of SREG XXXXXXXXX with specific value

Logged during XTR initialization phase

XTRNB: Skipping write to µCode patch MSR C0010020

Logged during XTR initialization phase

XTRNB: Processing GETMEMPTR request for XXXXXXXXXXX:...Denied

Logged during XTR execution phase where XXXXXX is the physical address of page requested. The request may be denied if it is requested for a MMIO region.

**** DEVMC_READMEM [800000007F1CAD00/296]: 55 8B EC 51 56 8B 75 0C**
**** DEVMC_WRITEMEM [400000007F294FD4/523]: A9 17 53 80**

Logged during XTR execution phase. 800000007F1CAD00 is the address 296 is the instruction count. The data following the ":" is the data that returned and received to and from the CPU. This message is logged for a READ/WRITE MEMORY request but no record is present in XTR XML file for this read. The data is hence served and written from and to backing store (whose contents were originally initialized from the XTR binary file)

XTRNB: Ir A03E w/event time = 326, Consume time = 597, CPU ICount = 99: 01 00
XTRNB: Iw A03E w/event time = 345, Consume time = 616, CPU ICount = 118: 00 00
XTRNB: Ia D1 w/event time = 326462, Consume time = 326462, CPU ICount = 326235

Logged during XTR execution phase when IOR/IOW message is received by XTRNB. A03E is the address of IOR/IOW and the data after the ":" is the data that is returned and received to and from the CPU. 'Ia' is for Interrupt Acknowledgement and D1 is the vector.

XTRNB: Time Resync - Adjusting time by -271...

Logged during execution when there is a timing discrepancy detected between an event in XTR XML and that received from the CPU. XTRNB adjusts to this discrepancy. In ideal environment this should not occur.

XTRNB: Queuing event CPU0[DMAW] for time 8403

Logged during execution when a DMAW event is queued so that it could be triggered at a later point. 8403 is the time when this event should be triggered.

XTRNB: Setting event trigger delay for CPU0[DMAW] to 1205
Logged during execution. DMAW event is setup to be triggered at a later point. 1205 is the difference between NOW and event time.

XTRNB: Processing queued event CPU0[DMAW] ICount=8403 ShellICount=8403.
Logged during execution. Trigger for event setup earlier is invoked. CPU0 and DMAW could have different values depending on which CPU it is (MP-XTR only) and which event is processed.

Interfaces
XTRNB has eight CPU interfaces and an IO Interrupt / APIC interface to connect to the AweSim's CPU Bus and IO Interrupt / APIC interface respectively. For XTR-UP, only one CPU interface may be used.

7.22.2 XTR Structure

7.22.2.1 XML Structure

XTR is a text file that contains XML elements for initialization elements, events and instructions. The XML schema or DTD is not formally defined. XTR XML contains an Initialization section followed by events and instruction sections. Last event in the XML must be an EOT event indicating the end of trace. Some XTR elements are explained below. Please refer to Section 7.22.5, "*Example XTR XML File*", on page 117, or the exact and complete structure of the XTR XML.

All values in the XML are in hexadecimal except for *ICount* and *Length* values which are always in decimal. Exceptions will be stated as necessary.

<Init Device="DIMM" Type="MEMI" Size="536870912" />
Memory initialization (MEMI) information from and for the DIMM device. The value for "Size" attribute the size of DIMM in bytes in decimal (base 10). Note that this does not require that XTR playback to have a DIMM device

<Init Device="MEM" Type="MEMI"
File="c:\simnow\xtr\DivergenceAt324303\test_snapshot_3dmarkwof_0.bin" />
Memory initialization file. File path may be relative to the current path.

<Init Device="CPU0" Type="CPU" Item="ICount" Data="227"/>
Initial instruction count in decimal. Different CPUs can have different initial ICounts.

<Init Device="CPU0" Type="CPU" Item="ModeFlags" Data="00000001"/>
The upper 32 bit of *ModeFlags* must contain Execution Control flags. Please refer to Section 7.22.3, "*ModeFlags*", on page 116 for more information.

<Init Device="CPU0" Type="SREG" Item="TSC" Data="0000000000000000"/>
The initialization information for MSRs. Note that initialization information for TSC will be ignored. Please use M00000010 for writes to TSC

<Init Device="CPU0" Type="APIC" Length="1024" >

```
.
.
.
</Init>
```

APIC initialization information.

```
<INSTR Device="CPU0" Type="FJMP" ICount="6778" JMP="1" RIP="f86b0619" />
```

An FJMP Instruction. RIP is optional and is only used to double check whether if the FJMP is taken at the correct instruction. JMP attribute can have the following values:

JMP=0: Force Do-not-take-jump for this instruction
JMP=1: Force Take-jump for this instruction

```
<Event Device="CPU0" Type="IOW" ICount="6817" Address="a038" Size="2">
<Data Length="2" Value="40af" />
</Event>
```

Defines an IOR or IOW dormant event.

```
<Event Device="CPU0" Type="DMAW" ICount="8403" Address="000000000c254340"
Length="64">
<Data Length="64"
Value="6d00005f5e5bc3909ac04600b7c04600d4c04600eec0460008c1460022c146003cc146
002fc2460067c2460085c24600a3c2460090909090909090909090909090" />
</Event>
```

Defines a DMAW event.

```
<Event Device="CPU0" Type="PIN" ICount="325496" Name="INTR" Level="A" />
```

Defines an INTR PIN event. Level="A" for *Asserted* or "D" for *Deasserted*. Name could be INTR, RESET, A20M, NMI, PAUSE, SMI, and <Unknown>.

```
<Event Device="TO_DO_IN_NB" Type="APIC" ICount="325496" Name="EXTINT"
DestinationMode="F" DeliveryMode="07" Level="F" TriggerMode="F" Vector="00"
Destination="00" />
```

Defines an APIC Event. Name could be EOI, INIT, STARTUP, SMI, NMI, INTR, REMOTE READ, EXTINT, LPARB, and Unknown. Device can be the name of the device that issues the interrupt. Current XTR implementation ignores the name of the device.

```
<Event Device="CPU0" Type="INTACK" ICount="325496" Vector="00000000000000d1" />
```

Defines an INTACK cycle event.

```
<Event Device="XTR" Type="EOT" ICount="400001" />
```

Defines an End of Trace (EOT) event.

```
<Event Device="CPU0" Type="RDMSR" ICount="1404861740" Address="00000010"
Data="0000000053BC7D2C" />
Defines a RDMSR event.

<Event Device="CPU0" Type="MEMR" ICount="3133971257"
Address="00000000000A88B2" Size="1">
        <Data Length="1" Value="FF" />
</Event>
```

```
<Event Device="CPU0" Type="MEMW" ICount="3133971259"
Address="00000000000A88B2" Size="1">
        <Data Length="1" Value="01" />
</Event>
```

Defines a Memory Read or Memory Write event. MEMR and MEMW are recorded for MMIO ranges.

7.22.2.2 XTR Binary File Contents

XTR Binary file contains the memory image of the system just before the XTR Record started. The binary file contains multiple records where each record contains has the following structure:

```
Physical Address Of the Page: 8 bytes
Count of Bytes in this Page: 4 Bytes
Data Of the Page: Count of Bytes earlier
```

Currently XTR only supports page size of 4096 bytes. Both the DIMM and MMIO may be present in the XTR Binary file. The last record in the binary file must have a count of zero to indicate end of memory image.

7.22.3 ModeFlags

ModeFlags defines some of the states of the CPU that are important for execution. The upper 32 bits store the Execution Control flags e.g. HLT and <ignore interrupts for 1 instruction when we change stack segment>. The lower 32 bits is redundant from other initialization values in the XTR initialization but is there to maintain code consistency.

Table 7-7 shows the Execution Control Flags (upper 32 bit):

Execution Control Flag	Value	Description
BIUI_LOCK	0x00000001	Bus is locked
BIUI_RESET	0x00000002	Processor RESET pin.
BIUI_INIT	0x00000004	INIT pin
BIUI_INTR	0x00000008	Interrupt
BIUI_NMI	0x00000010	NMI
BIUI_SMI	0x00000020	SMI
BIUI_IGNNE	0x00000040	Floating point IGNNE
BIUI_A20M	0x00000080	A20Mask
BIUI_PAUSE	0x00000100	PAUSE
BIUI_HOLD	0x00000200	HOLD
BIUI_UNUSED	0x00000400	Unused
BIUI_STOP	0x00000800	Pseudo pin that stops simulation

Table 7-7: Execution Control Flags

Table 7-8 shows other internal execution control flags. Some flags may be AweSim specific.

Execution Control Flag	Value	Description
ECF_SMCRESTART	0x00001000	SMC detected in current translation (restart required).
ECF_GENEXCEPTION	0x00002000	SVM virtual interrupt pending
ECF_VINTR	0x00004000	INIT pin

Execution Control Flag	Value	Description
ECF_UNUSED	0x00008000	Unused
ECF_HALT	0x00010000	We are in a HALT
ECF_SHUTDOWN	0x00020000	We are in a SHUTDOWN
ECF_FPUHANG	0x00040000	FPU freeze
ECF_APICHOLD	0x00080000	APIC freeze
ECF_IGNOREINTR	0x00100000	Ignore INTR for one instruction
ECF_TRAP	0x00200000	EFlags.TF bit
ECF_EXECBP	0x00400000	User execution breakpoints exist
ECF_LATCHEDSMI	0x00800000	A latched SMI was seen
ECF_STACKEDSMI	0x01000000	A latched SMI from within an SMI
ECF_LATCHEDNMI	0x02000000	A latched NMI was seen
ECF_SMIEDGE	0x04000000	An SMI edge has been detected
ECF_NMIEDGE	0x08000000	An NMI edge has been detected
ECF_APICMSGPENDING	0x10000000	An APIC message is waiting to be handled
ECF_APICACTPENDING	0x20000000	Any other APIC activity is pending
ECF_DR7CODEBREAKS	0x40000000	DR7 has code breakpoints enabled
ECF_LASTWASIO	0x80000000	Set if previous. instruction did I/O

Table 7-8: Internal Execution Control Flags

7.22.4 Limitations

- Any line in XTR XML file cannot be greater than 255 characters.
- Comment start tag "<!--" should start on a new line and end tag "-->" should be last characters on a line.
- The XML attributes are case sensitive but the values are not.
- XTR cannot be used to playback BIOS bring-ups.
- Currently XTR does not support Pacifica platform.
- Currently XTR traces recorded off SimNow cannot be played back in other XTR playback environments.
- Although not needed, XTR traces recorded by SimNow might contain data written by the CPU, e.g. IOW.

7.22.5 Example XTR XML File

```
<?xml version="1.0" encoding="utf-8" ?>
<AmdEventTrace version="1.0">
<Init Device="DIMM" Type="MEMI" Size="536870912" />
<Init Device="MEM" Type="MEMI" File="xtr1.bin" />
<Init Device="CPU0" Type="CPU" Item="ICount" Data="227" />
<Init Device="CPU0" Type="CPU" Item="RIP" Data="0000000082D6A8E4" />
<Init Device="CPU0" Type="CPU" Item="RAX" Data="0000000000628E01" />
<Init Device="CPU0" Type="CPU" Item="RBX" Data="000000000B0BE41C" />
<Init Device="CPU0" Type="CPU" Item="RCX" Data="000000000B080E20" />
<Init Device="CPU0" Type="CPU" Item="RDX" Data="0000000000000080" />
<Init Device="CPU0" Type="CPU" Item="RSI" Data="0000000000C8FA38" />
<Init Device="CPU0" Type="CPU" Item="RDI" Data="000000000B09A6B8" />
<Init Device="CPU0" Type="CPU" Item="RBP" Data="000000000B0BEFE0" />
<Init Device="CPU0" Type="CPU" Item="RSP" Data="00000000B043ADCC" />
<Init Device="CPU0" Type="CPU" Item="R8" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="R9" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="R10" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="R11" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="R12" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="R13" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="R14" Data="0000000000000000" />
```

```
<Init Device="CPU0" Type="CPU" Item="R15" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="ModeFlags" Data="00000001" />
<Init Device="CPU0" Type="CPU" Item="EFlags" Data="0000000000000002" />
<Init Device="CPU0" Type="CPU" Item="ES" Data="00000023" />
<Init Device="CPU0" Type="CPU" Item="ESBase" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="ESLimit" Data="00000000FFFFFFFF" />
<Init Device="CPU0" Type="CPU" Item="ESFlags" Data="00000CF3" />
<Init Device="CPU0" Type="CPU" Item="CS" Data="00000008" />
<Init Device="CPU0" Type="CPU" Item="CSBase" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="CSLimit" Data="00000000FFFFFFFF" />
<Init Device="CPU0" Type="CPU" Item="CSFlags" Data="00000C9B" />
<Init Device="CPU0" Type="CPU" Item="SS" Data="00000010" />
<Init Device="CPU0" Type="CPU" Item="SSBase" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="SSLimit" Data="00000000FFFFFFFF" />
<Init Device="CPU0" Type="CPU" Item="SSFlags" Data="00000C93" />
<Init Device="CPU0" Type="CPU" Item="DS" Data="00000023" />
<Init Device="CPU0" Type="CPU" Item="DSBase" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="DSLimit" Data="00000000FFFFFFFF" />
<Init Device="CPU0" Type="CPU" Item="DSFlags" Data="00000CF3" />
<Init Device="CPU0" Type="CPU" Item="FS" Data="00000038" />
<Init Device="CPU0" Type="CPU" Item="FSBase" Data="000000007FFDE000" />
<Init Device="CPU0" Type="CPU" Item="FSLimit" Data="0000000000000FFF" />
<Init Device="CPU0" Type="CPU" Item="FSFlags" Data="000004F3" />
<Init Device="CPU0" Type="CPU" Item="GS" Data="00000000" />
<Init Device="CPU0" Type="CPU" Item="GSBase" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="GSLimit" Data="000000000000FFFF" />
<Init Device="CPU0" Type="CPU" Item="GSFlags" Data="00000000" />
<Init Device="CPU0" Type="CPU" Item="LDTR" Data="00000000" />
<Init Device="CPU0" Type="CPU" Item="LDTBase" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="LDTLimit" Data="000000000000FFFF" />
<Init Device="CPU0" Type="CPU" Item="LDTFlags" Data="00000000" />
<Init Device="CPU0" Type="CPU" Item="TR" Data="00000028" />
<Init Device="CPU0" Type="CPU" Item="TSSBase" Data="0000000080042000" />
<Init Device="CPU0" Type="CPU" Item="TSSLimit" Data="00000000000020AB" />
<Init Device="CPU0" Type="CPU" Item="TSSFlags" Data="00000089" />
<Init Device="CPU0" Type="CPU" Item="IDTBase" Data="000000008003F400" />
<Init Device="CPU0" Type="CPU" Item="IDTLimit" Data="00000000000007FF" />
<Init Device="CPU0" Type="CPU" Item="GDTBase" Data="000000008003F000" />
<Init Device="CPU0" Type="CPU" Item="GDTLimit" Data="00000000000003FF" />
<Init Device="CPU0" Type="CPU" Item="DR0" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="DR1" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="DR2" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="DR3" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="DR6" Data="00000000FFFF0FF0" />
<Init Device="CPU0" Type="CPU" Item="DR7" Data="0000000000000400" />
<Init Device="CPU0" Type="CPU" Item="CR0" Data="0000000080010031" />
<Init Device="CPU0" Type="CPU" Item="CR2" Data="000000000000000C" />
<Init Device="CPU0" Type="CPU" Item="CR3" Data="000000000043D000" />
<Init Device="CPU0" Type="CPU" Item="CR4" Data="00000000000006D9" />
<Init Device="CPU0" Type="CPU" Item="CR8" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="TSC" Data="00000000000000E3" />
<Init Device="CPU0" Type="SREG" Item="M00000010" Data="00000000000000E3" />
<Init Device="CPU0" Type="SREG" Item="MC0010111" Data="0000000001000000" />
<Init Device="CPU0" Type="SREG" Item="MC0000080" Data="00000000" />
<Init Device="CPU0" Type="SREG" Item="MC0000100" Data="000000007FFDE000" />
<Init Device="CPU0" Type="SREG" Item="MC0000101" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0000102" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0011004" Data="000000008001350C" />
<Init Device="CPU0" Type="SREG" Item="M000000FE" Data="0000000000000508" />
<Init Device="CPU0" Type="CPU" Item="FCW" Data="0000107F" />
<Init Device="CPU0" Type="CPU" Item="FSW" Data="00000020" />
<Init Device="CPU0" Type="CPU" Item="FTW" Data="0000FFFF" />
<Init Device="CPU0" Type="CPU" Item="FDS" Data="00000000" />
<Init Device="CPU0" Type="CPU" Item="FCS" Data="00000000" />
<Init Device="CPU0" Type="CPU" Item="FIP" Data="0000000000000000" />
<Init Device="CPU0" Type="CPU" Item="FOP" Data="00000000" />
```

```
<Init Device="CPU0" Type="CPU" Item="FP0" Data="00000000000000000000" />
<Init Device="CPU0" Type="CPU" Item="FP1" Data="00000000000000000000" />
<Init Device="CPU0" Type="CPU" Item="FP2" Data="3ffee6455d0000000000" />
<Init Device="CPU0" Type="CPU" Item="FP3" Data="3ffdb139430000000000" />
<Init Device="CPU0" Type="CPU" Item="FP4" Data="4005c45c6d0000000000" />
<Init Device="CPU0" Type="CPU" Item="FP5" Data="4004ccf8aa0000000000" />
<Init Device="CPU0" Type="CPU" Item="FP6" Data="40018ac7100000000000" />
<Init Device="CPU0" Type="CPU" Item="FP7" Data="40068d00470000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0000081" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0000082" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0000083" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000001B" Data="00000000FEE00900" />
<Init Device="CPU0" Type="SREG" Item="M00000200" Data="0000000000000006" />
<Init Device="CPU0" Type="SREG" Item="M00000202" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000204" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000206" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000208" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000020A" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000020C" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000020E" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000201" Data="000000FF80000800" />
<Init Device="CPU0" Type="SREG" Item="M00000203" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000205" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000207" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000209" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000020B" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000020D" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000020F" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000250" Data="1E1E1E1E1E1E1E1E" />
<Init Device="CPU0" Type="SREG" Item="M00000258" Data="1E1E1E1E1E1E1E1E" />
<Init Device="CPU0" Type="SREG" Item="M00000259" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000268" Data="1515151515151515" />
<Init Device="CPU0" Type="SREG" Item="M00000269" Data="1010101010101010" />
<Init Device="CPU0" Type="SREG" Item="M0000026A" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000026B" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000026C" Data="0404040404040404" />
<Init Device="CPU0" Type="SREG" Item="M0000026D" Data="0404040404040404" />
<Init Device="CPU0" Type="SREG" Item="M0000026E" Data="1010101010101010" />
<Init Device="CPU0" Type="SREG" Item="M0000026F" Data="1010101010101010" />
<Init Device="CPU0" Type="SREG" Item="M000002FF" Data="0000000000000C00" />
<Init Device="CPU0" Type="SREG" Item="M00000400" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000405" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000408" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M0000040C" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000410" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M000001D9" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M000001DB" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M000001DC" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M000001DD" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M000001DE" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000277" Data="0007010600070106" />
<Init Device="CPU0" Type="SREG" Item="M00000174" Data="0000000000000008" />
<Init Device="CPU0" Type="SREG" Item="M00000175" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="M00000176" Data="000000008052D480" />
<Init Device="CPU0" Type="CPU" Item="MXCSR" Data="0000000000001F80" />
<Init Device="CPU0" Type="CPU" Item="XMM00" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM00" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM01" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM02" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM03" Data="00000000000000000000000000000000"
/>
```

```
<Init Device="CPU0" Type="CPU" Item="XMM04" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM05" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM06" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM07" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM08" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM09" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM10" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM11" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM12" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM13" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM14" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="CPU" Item="XMM15" Data="00000000000000000000000000000000"
/>
<Init Device="CPU0" Type="SREG" Item="MC0010010" Data="0000000000160601" />
<Init Device="CPU0" Type="SREG" Item="MC0010015" Data="000000000A000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010016" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010017" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010018" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010019" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC001001A" Data="0000000080000000" />
<Init Device="CPU0" Type="SREG" Item="MC001001D" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010030" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010031" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010032" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010033" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010034" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010035" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010112" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0010113" Data="0000000000000001" />
<Init Device="CPU0" Type="SREG" Item="MC0011020" Data="0000000000000000" />
<Init Device="CPU0" Type="SREG" Item="MC0011023" Data="0000000000000000" />
<Init Device="CPU0" Type="APIC" Length="1024">
<Data Length="16" Value="00000000000000000000000010000400" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="000000000000000ffffffffff000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000010000000000" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000000000000000000000000000000" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000100000001000000010000000100" />
<Data Length="16" Value="00000100000001000000010000000100" />
```

```
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
<Data Length="16" Value="0000010000000100000001000000100" />
</Init>
<!-- Trace started on instruction 227 -->
<Event Device="CPU0" Type="IOR" ICount="326" Address="a03e" Size="2">
        <Data Length="2" Value="0100" />
</Event>
<Event Device="CPU0" Type="IOW" ICount="345" Address="a03c" Size="2">
        <Data Length="2" Value="0000" />
</Event>
<Event Device="CPU0" Type="IOW" ICount="364" Address="a03e" Size="2">
        <Data Length="2" Value="1100" />
</Event>
<Event Device="CPU0" Type="IOR" ICount="588" Address="a037" Size="1">
        <Data Length="1" Value="0c" />
</Event>
<INSTR Device="CPU0" Type="FJMP" ICount="6778" JMP="1" RIP="f86b0619" />
<INSTR Device="CPU0" Type="FJMP" ICount="6797" JMP="1" RIP="f86b0619" />
<Event Device="CPU0" Type="IOW" ICount="6817" Address="a038" Size="2">
        <Data Length="2" Value="40af" />
</Event>
<INSTR Device="CPU0" Type="FJMP" ICount="7081" JMP="1" RIP="f86b0317" />
<INSTR Device="CPU0" Type="FJMP" ICount="7099" JMP="1" RIP="f86b0317" />
<Event Device="CPU0" Type="IOR" ICount="7110" Address="a037" Size="1">
        <Data Length="1" Value="0d" />
</Event>
<Event Device="CPU0" Type="IOR" ICount="7121" Address="a037" Size="1">
```

```
        <Data Length="1" Value="0d" />
</Event>
<Event Device="CPU0" Type="IOR" ICount="7137" Address="a03e" Size="2">
        <Data Length="2" Value="0000" />
</Event>
<Event Device="CPU0" Type="IOW" ICount="7198" Address="a03c" Size="2">
        <Data Length="2" Value="5fc0" />
</Event>
<Event Device="CPU0" Type="DMAW" ICount="8403" Address="000000000c254340" Length="64">
        <Data Length="64"
Value="6d00005f5e5bc3909ac04600b7c04600d4c04600eec0460008c1460022c146003cc14600
2fc2460067c2460085c24600a3c2460090909090909090909090909090" />
</Event>
<Event Device="CPU0" Type="DMAW" ICount="18228" Address="000000000e67dc00"
Length="64">
        <Data Length="64"
Value="00005f5e5d5b64890d0000000081c414040000c218008bff293b47003b3b47003b3b4700
4d3b47004d3b47004d3b4700568bf18b460c85c0c706f4eb5b007406" />
</Event>
<Event Device="CPU0" Type="DMAW" ICount="23921" Address="000000000c254340"
Length="64">
        <Data Length="64"
Value="6d00005f5e5bc3909ac04600b7c04600d4c04600eec0460008c1460022c146003cc14600
2fc2460067c2460085c24600a3c2460090909090909090909090909090" />
</Event>
<Event Device="CPU0" Type="PIN" ICount="326462" Name="INTR" Level="A" />
<Event Device="TO_DO_IN_NB" Type="APIC" ICount="326462" Name="EXTINT"
DestinationMode="F" DeliveryMode="07" Level="F" TriggerMode="F" Vector="00" Destination="00" />
<Event Device="CPU0" Type="PIN" ICount="326462" Name="INTR" Level="D" />
<Event Device="CPU0" Type="INTACK" ICount="326462" Vector="00000000000000d1" />
<Event Device="CPU0" Type="IOW" ICount="326532" Address="70" Size="1">
        <Data Length="1" Value="0c" />
</Event>
<Event Device="CPU0" Type="IOR" ICount="326536" Address="71" Size="1">
        <Data Length="1" Value="c0" />
</Event>
<Event Device="CPU0" Type="IOW" ICount="326541" Address="70" Size="1">
        <Data Length="1" Value="0c" />
</Event>
<Event Device="CPU0" Type="IOR" ICount="326545" Address="71" Size="1">
        <Data Length="1" Value="00" />
</Event>
<Event Device="XTR" Type="EOT" ICount="400967" />
</AmdEventTrace>
```

7.23 JumpDrive Device

The purpose of the JumpDrive device is to allow easy import and export of data between a host system and a simulation environment. You can import files from the host system on to the JumpDrive, where they will be accessible by the simulated operating system. Data can also be exported from the JumpDrive back to the host system after the simulation ended.

The image file used by the JumpDrive is very different from any other image files that the simulator supports. The only image files that can be loaded are those image files that are saved by the JumpDrive itself.

Section A.7.26, “JumpDrive”, on page 244 describes the JumpDrives automation commands.

Interface
The JumpDrive device has an USB interface that can connect to any USB controller, e.g., you can connect the JumpDrive device to the AMD-8111 I/O Hub.

Initialization and Reset State
The JumpDrives initialized state is all zero. There is no partition table or any other structure defined. It is totally blank. The default size is 64 Mbytes. The JumpDrive is not modified after a reset.

Contents of a BSD
The JumpDrive device saves its entire state, including the contents of its memory, to the BSD. Any data that exists on the JumpDrive device will be restored when the BSD is reloaded.

Configuration Options
Most of the automation commands will return an error if the JumpDrive is "plugged into" the simulated computer, i.e., if the JumpDrive device is connected to a USB controller. The device must be "not connected", i.e., unplugged, to issue commands that alter the JumpDrive image.

7.24 E1000 Network Adapter Device

The network adapter device models an Intel Pro/1000 MT Desktop Network Adapter. The adapter depends heavily on MAC address assignment in order to determine how visible it is to *real* network resources or other simulator network sessions. The adapter model requires a separate mediator process to bridge access to the real network. This device provides a list of automation commands that can be used to configure the adapter model, see Section A.7, "*Automation Commands*", on page 224.

To model network workloads the following are typically required:

1. One or more BSDs with a NIC device included in each BSD.
2. A mediator process running remotely or locally.

The mediator is a background daemon task, whose purpose is to bridge the NIC model to the real network or other SimNow BSDs. The level of network visibility for each simulator session depends on the format of the MAC address that is used for the simulated NIC model.

Figure 7-32 shows depicts four simulator sessions communicating via a mediator.

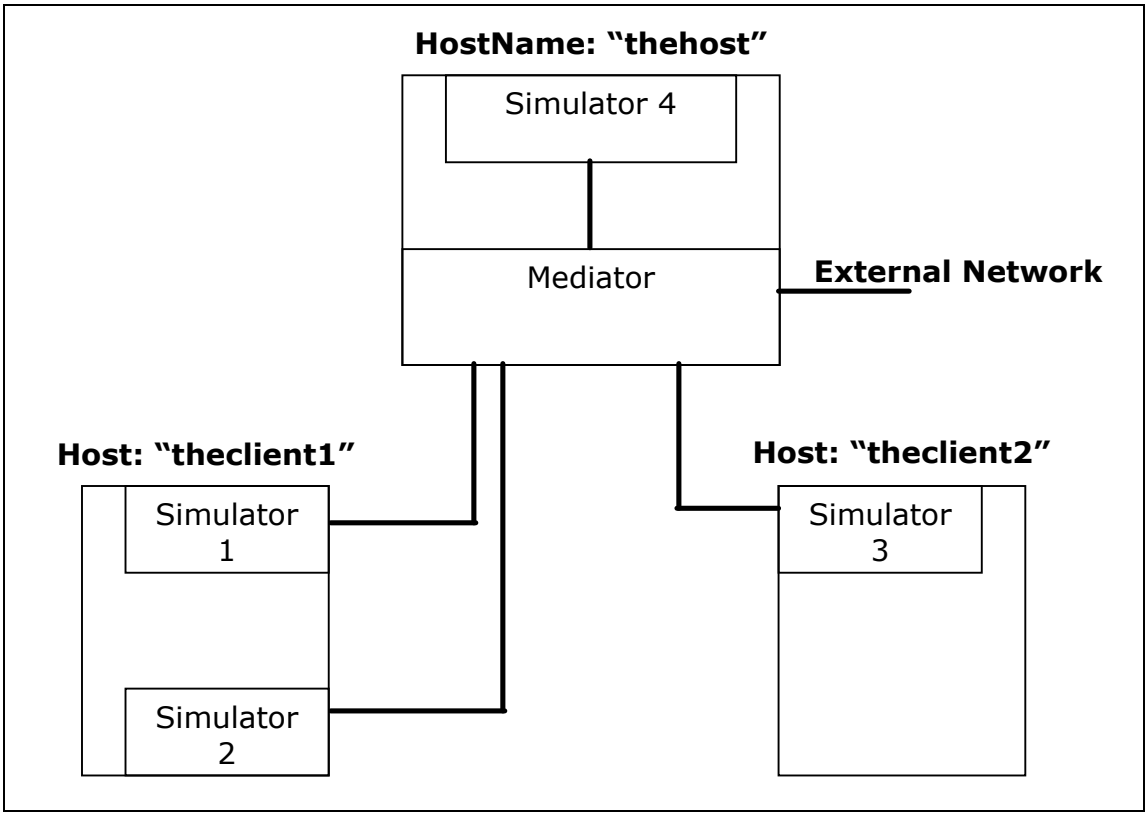


Figure 7-32: Communication via Mediator

Alternatively a multi-machine approach can be used in which multiple BSD's are loaded in the same process space. This architecture allows the simulator sessions to pass packets back and forth without the need for a mediator. Running without a mediator isolates the simulator sessions from the real network. For more information on running multiple simulator instances in the same process, see Section 5.3, *Multi-Machine Support*, on page 45.

Figure 7-33 illustrates multi-machine communication of simulator sessions without a mediator.

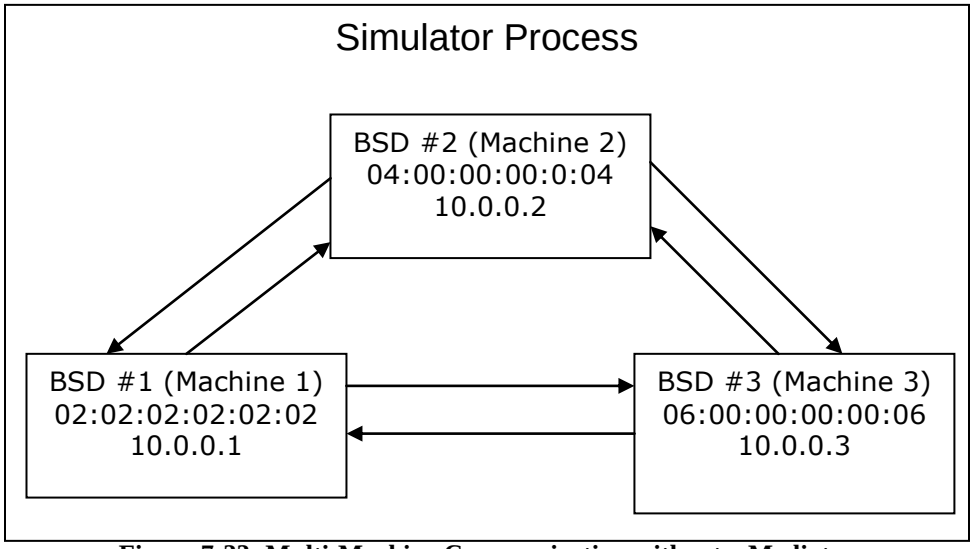


Figure 7-33: Multi-Machine Communication without a Mediator

7.24.1 Simulated Link Negotiation

A link will appear connected in the guest system when one of the following occurs:

- A mediator connection has been established.
- There is at least one other NIC BSD running in the same process, and are aware of each other.

When a new mediator connection string has been specified, a one-shot link negotiation will take place within the simulator. Depending on whether a connection was made with the mediator, the link will appear to be connected or disconnected on the guest. If the mediator was killed and has since been restarted, then the user will need to perform a "linkConnect auto", to restart link negotiation.

Similarly, in a multi-machine setup, the first simulator session will also need to perform a "linkConnect auto" to ensure that the initial guest sees that other simulator peers have been connected.

When neither of the above conditions is met, the link appears disconnected in the guest. It may be necessary to re-start link negotiation via "linkConnect auto". This will cause

the NIC model to retry a mediator connection or search for any simulator peers, running in the same process.

7.24.2 The Mediator Daemon

The mediator provides several services for the simulator session:

- Access to real network resources (DHCP servers, etc.). Note that the mediator will need to be run with supervisor privileges in order to snoop network traffic on its host.
- Bridge communication to other simulator sessions.
- Group individual sessions into domains so that identical BSD's (with identical MAC/IP pairs) can be run simultaneously in separate domains.
- Provides an optional gateway to block broadcast traffic and to perform Network Address Translation (NAT) on identical BSD's in different domains.

The mediator can route traffic to and from the real network. This operation requires low-level kernel actions, so the mediator must be run by a supervisor with sufficient OS privileges. Users may want to have one machine on the subnet dedicated to running the mediator in this mode. Client machines that connect to the mediator will not require supervisor privileges.

The mediator is capable of grouping certain simulator sessions into *domains*. Domains isolate groups of simulator sessions from each other. This can be useful when the user wants to run replicated groups of BSD's simultaneously. The user need to ensure that each group of BSD's are using unique domains in the mediator by passing an appropriate connect string to the mediator or supplying it on the command line using the "-m" option, see Section 5.1, *Command-Line Arguments*, on page 39.

The mediator can provide one or more gateways to isolate broadcast traffic from your simulation environment. A gateway will perform NAT in order to ensure that BSD's in different domains get their packets routed appropriately. The simulator sessions using the mediator's gateway can continue to access network resources, but are essentially hidden from the real network.

Table 7-9 shows command line switches that the mediator accepts:

Switch	Description
-p portNum	Dictates what port number the mediator will be listening on for incoming traffic. It specifies the base port address used by the mediator, and port usage is based off of this number. The mediator's listening thread uses portNum + 4.
-l	Lists possible host adapters that the mediator can use to snoop real network traffic.
-s	Tells the mediator to snoop real network traffic. Requires supervisor privileges.
-d DeviceNum	Tells the mediator which host adapter to use when snooping real

	network traffic. This device number will need to be one listed using the "-l" command.
-v[v][v]	Turns on verbose output. The verbosity level gets noisier with the number of "v" on the command line.
-m XX:XX	Denotes the two high bytes used to classify the simulator's MAC addresses. By default these values are FA:CD, but can be configured to avoid collisions with real hardware.

Table 7-9: Mediator Command Line Switches

7.24.3 MAC Addresses for use with the Adapter

The MAC address that the simulated adapter is using determines the level of visibility that the model will have with other simulator sessions and with the real network. The mediator routes packets to simulator sessions that have "FA:CD" in the high two bytes of the MAC address. The simulator sessions that have anything other than "FA:CD" can only communicate with other simulator sessions in the same process space using a "multi-machine" approach.

MAC Address beginning with "FA:CD" and having a third byte between 0x00 and 0x20, are classified as "absolute". Simulated adapters using this class of MAC Address are logically equivalent to plugging a real computer into a real network. These sessions can see real network traffic and are visible to all simulator sessions running via the mediator. In addition, all broadcast traffic, including ARP's are routed to this class of MAC addresses. Allocations of "absolute" MAC addresses need to be coordinated such that they are not replicated on the same host subnet.

MAC addresses beginning with "FA:CD" and having a third byte between 0x21 and 0x80 are classified as "fixed". The simulator adapters using this class of MAC address can access the real network, but cannot be seen by other simulator sessions outside of its domain. This class of MAC address allows a user to simultaneously run identical BSD's using unique domains. This class of MAC addresses will not receive broadcast traffic such as ARP's. Allocations of "fixed" MAC addresses need to be coordinated such that they are not replicated in the same mediator domain.

7.24.4 Example Configurations

MAC address assignment was designed to satisfy many usability needs. Table 7-10 shows a list of possible usage models for the simulator and MAC Address assignments.

7.24.4.1 Absolute NIC

This configuration mimics plugging in a physical computer into whatever network your mediator is running on. The user must select a MAC Address that is not duplicated anywhere else on the mediator's subnet. All broadcast and targeted network traffic will be routed to a simulator session using this classification of MAC Address. This provides maximum visibility for the simulator session.

Example MAC:	FA:CD:00:00:00:01
IP Address:	Any. Can be a static IP address assigned by your sys admin, or a

	DHCP acquired address.
Visibility:	Can be seen by external network and all simulator sessions running anywhere on the network.
Mediator String:	"Hostname"

Table 7-10: MAC Address Assignments

7.24.4.2 Client-Server simulated network

This configuration uses "fixed" MAC addresses to allow this domain to be replicated in the mediator space, without colliding with one another. To allow real network access, we will also run the mediator with a gateway at IP address 192.168.0.1.

Example MAC:	FA:CD:21:00:00:01
IP Address:	Static IP address 192.168.0.2
Visibility:	Accesses the real network via the mediator's gateway. External network hosts can not directly communicate with this client.
Mediator String:	mydomain@hostname

Table 7-11: Client-Server: Simulator Server

Example MAC:	FA:CD:22:00:00:02
IP Address:	Static IP address 192.168.0.3
Visibility:	Accesses the real network via the mediator's gateway. External network hosts can not directly communicate with this client.
Mediator String:	mydomain@hostname

Table 7-12: Client-Server: Simulator Client 1

The BSD's that contain the server and client can be run simultaneously on the same network without any collisions. They will require the user to input different domains in the mediator connection string, see also Section 5.1, *Command-Line Arguments*, on page 39 (-m option).

7.24.4.3 Isolated Client-Server simulated network (Same process)

This type of setup isolates the simulator sessions from the real network, only allowing visibility to other simulator sessions in the same process. A mediator is not required for this type of setup.

Example MAC:	02:00::00:00:00:01
IP Address:	Static IP address 192.168.0.1
Visibility:	Can only communicate with BSD's in the same simulator process using multiple machines.
Mediator String:	N/A

Table 7-13: Isolated Client-Server: Simulator Server

Example MAC:	02:00::00:00:00:02
IP Address:	Static IP address 192.168.0.2

Visibility:	Can only communicate with BSD's in the same simulator process using multiple machines.
Mediator String:	N/A

Table 7-14: Isolated Client-Server: Simulator Client 1

7.24.5 Visibility Diagram

Figure 7-34 depicts the mediator routing packets to and from several simulator sessions in different domains. The session running BSD #3 is using an absolute MAC address, and therefore is globally visible. This session is no different than any other machine running on the external network. All simulator sessions, connected to any mediator, will be able to see this machine.

Notice also that domains one and two are using identical BSDs that are running simultaneously. To prevent collisions on the external network, the mediator will not route broadcast packets to these sessions as they are using a fixed MAC classification. The gateway will be able to do network address translation (NAT) for each BSD in each domain to make sure that there are no collisions between the two domains.

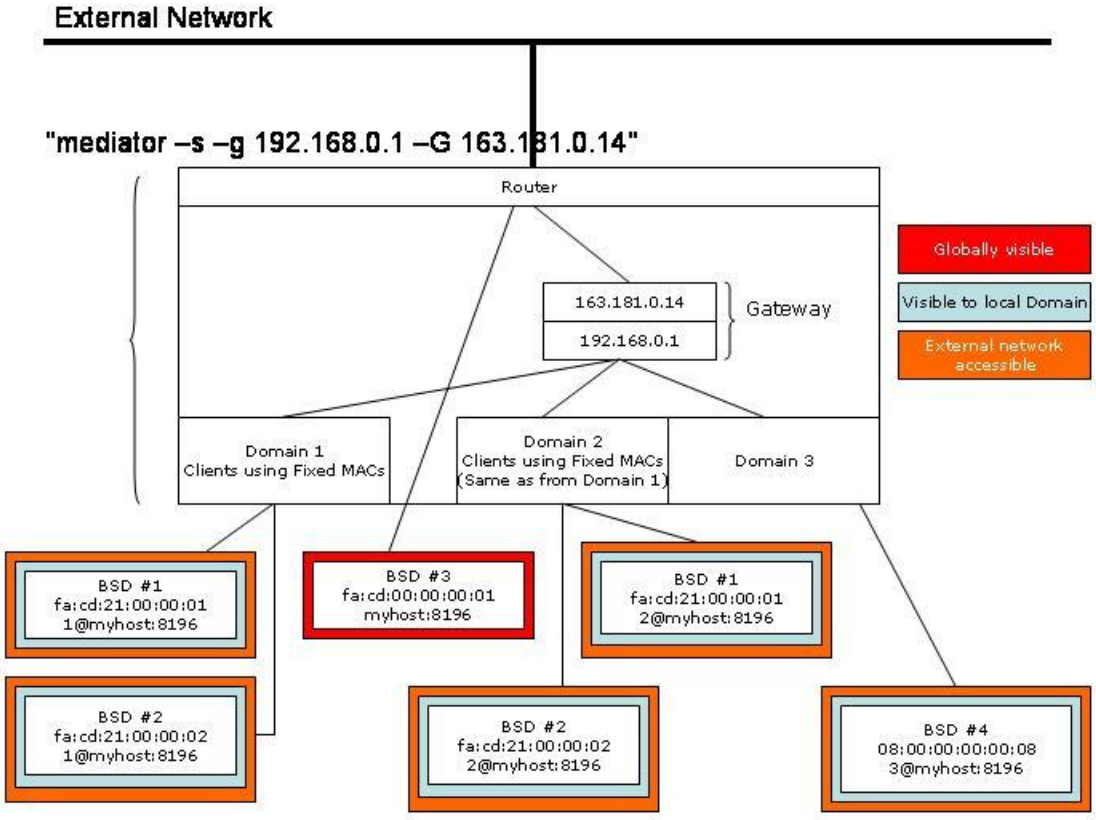


Figure 7-34: Visibility Diagram

7.25 Plug and Play Monitor Device

The Plug and Play Monitor device (PnP Monitor) conforms to the VESA Plug and Play Monitor specification and therefore supports the DDC2B standard. DDC (**D**isplay **D**ata **C**hannel) is the Plug and Play standard for monitors. DDC monitors are designed to meet the VESA (**V**ideo **E**lectronic **S**tandards **A**ssociation) standard that defines the DDC implementation. If the video card also supports the DDC standard it gets from the PnP monitor device all the information about its features and makes consequently an automatic configuration for the best refresh values depending on the selected resolution.

The Plug and Play monitor device supports the DDC1 and DDC2B standards. DDC1 is primitive and a point to point interface. The monitor is always put at transmit-only mode (DDC1). The monitor will continuously transmit data until the monitor will be turned off or switched to the bi-directional mode (DDC2). In DDC2 mode the I^2C protocol is being used for data transfers.

Interface
The Plug and Play Monitor device model has a VGA and DVI interface connection. Connections can be only made to the VGA or DVI interface. It can be connected to the VGA or DVI connection of a video card device.

Contents of a BSD
The current state of all internal registers and any internal state variables are saved in the BSD.

Initialization and Reset State
When first initialized or reset the Plug and Play Monitors DDC registers are set to their default state. After initialization the monitor device will operate in DDC1 mode. The device will remain in the DDC1 mode until there is a valid HIGH to LOW transition on the SCL pin, when it will switch to DDC2B mode.

Differences from Real Hardware
The model attempts to match the functionality of the physical devices from a programmer's perspective. Upon power-up, a “real” Plug and Play monitor will output valid data only after it has been initialized. During initialization, data will not be available until after the first nine clock cycles are sent to the device. This Plug and Play monitor device model does not simulate this behaviour. It will always output valid data.

The *Page Write*, *Acknowedge Polling*, and the *Write Protection* feature are currently not supported.

Configuration Options
The Plug and Play Monitor device gives you the opportunity to choose from different Plug and Play Monitor device models, as shown in Figure 7-35.

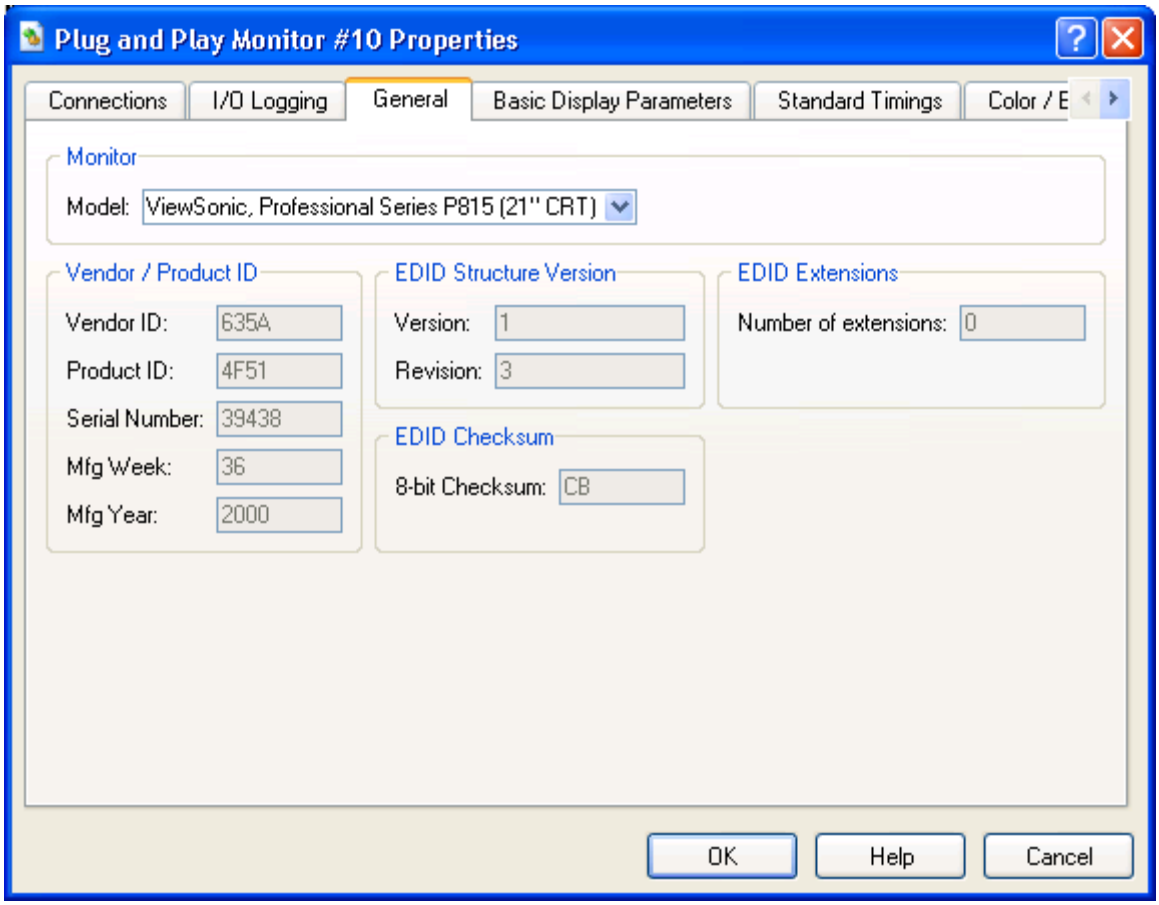


Figure 7-35: Plug and Play Monitor Device Configuration

7.26 ATI SB400/SB600/SB700 Southbridge Devices

The ATI Southbridge devices provide the basic I/O Southbridge functionality of the system. Features include 4 or 6 SATA ports, a PIO-mode IDE controller supporting 1 or 2 channels, fully functinoal USB 1.1 Controller supporting legacy emulation, an LPC/ISA bridge, an SMB 2.0 compliant controller, an IOAPIC controller, HPET timer, and legacy AT devices (8259 PIC, 8254 PIT, CMOS, and DMA controller). The legacy AT devices have the standard behavior and IO addresses unless otherwise noted.

Interface

The Southbridge devices have several connection points. Possible connection points include a PCI bus, an SMB bus, an LPC bus, and an upstream PCI-E link. The PCI bus acts as a host bus, and should connect to a "PCI Bus Device". The SMB connects to devices such as the DIMM, an SMB hub device, or another SMB compatible endpoint. The LPC bus provides connectivity to devices such as Super IO chips and BIOS ROMs. The PCI-E port is used for connectivity upstream to a compatible Northbridge Device. See Section 7.27, "*ATI RS480/RS780/RD790/RD890 Northbridge Devices*", on page 134 for more information.

Initialization and Reset State

When first initialized, the Southbridge devices are in the default state. This is described in detail in the respective datasheets. The legacy CMOS sub device initializes to all zeroes.

When reset, a Southbridge device takes on all default register values as above. The exception to this is that the CMOS contents remain the same.

Contents of a BSD

The BSD file contains the contents of all registers. It also saves the contents of any buffers, and states of all internal devices (HDD controllers, PIT, PIC, etc.). When the BSD file is read in, all buffers are filled with past data, and all states are restored to their saved states.

Configuration Options

These Southbridge devices share many configuration properties with the AMD-8111 Southbridge. For more information please refer to Section 7.11, "*AMD-8111™ Southbridge Devices – IO Hubs*", on page 90.

Addittionaly these SouthBridge devices contain a SATA configuration page to attatch images to the individual SATA ports.

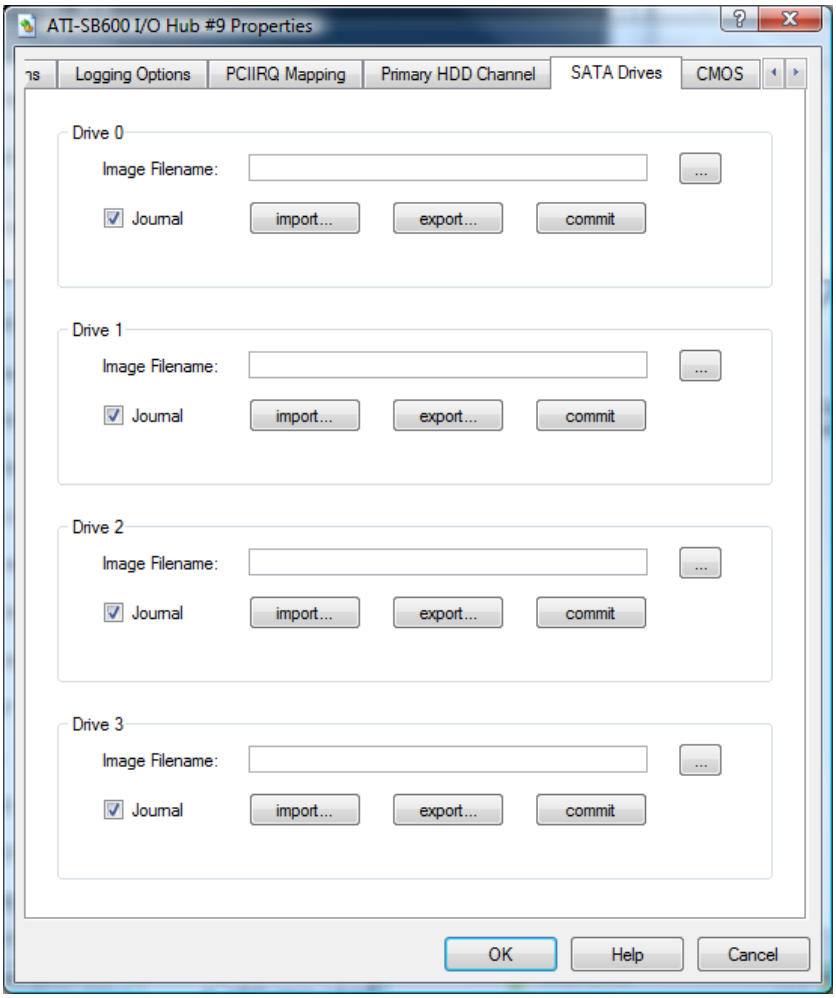


Figure 7-36: ATI SB600 SATA Configuration Dialog

Log Messages

These SouthBridge devices have the ability to log messages to the Message Log Window as specified by the options in the Logging Option tab. These devices can log I/O-mapped Transactions, Memory-mapped Transactions, and SMI and SCI assertions.

Difference from Real Hardware

These Southbridge devices differ from other devices mainly in those items that deal with real-time operation. Those items cannot be modeled in the current simulator. The functionality of the USB 2.0 controller is also absent (PCI registers and memory-mapped registers are the only portion present). Hardware supporting HD Audio is also not modelled in SimNow.

7.27 ATI RS480/RS780/RD790/RD890 Northbridge Devices

The ATI RS480/RD790/RS780 feature set includes an upstream HyperTransport CPU interface, a PCI-E interface, and an A-Link PCI-E dowstream interface to the SouthBridge. Depending on the part and the platform, each device may have some number of available PCI-E slots to connect with endpoint devices.

Interface
These Northbridge devices provide an upstream HyperTransport interface for communication with the Host. The Downstream link is a 2x or 4x PCI-E link used for communication with a SouthBridge device. Several PCI-E slot interfaces are also available. The number of slots varies by part and platform specifications.

Contents of a BSD
The current state of all PCI configuration registers and any internal state variables are saved in the BSD.

Configuration Options
No configuration options currently.

Log Messages
No logging is provided, other than the global options provided by each device. See Section 9.3, "*I/O Logging*", on page 144 for more information.

Difference from Real Hardware
The ATI RS480 and ATI RS780 device models do not simulate their integrated graphics processors. The RS780 model does not simulate the integrated HD Audio device.

7.28 AMD "Istanbul" Device

The AMD "*Istnabul*" device is a 6 core processor node, suitable for an L1 socket. It emulates a planned product that derives from a revision of the AMD Family10h product line. The device iteself is composed of 6 individual *AweSim Processor Devices* that are connected to a single "*AMD 8th Generation Integrated Northbridge Device*".

For more information on Group Devices, see Section 3.3, "*Device Groups*", on page 3.3.

Interface
AMD "*Istanbul*" Device has several connection ports. It has 4 HyperTransport links split to form 8 sub-links. Each sub-link can connect to a coherent HyperTransport device (such as another *AMD "Istanbul" Device*) or a non-Coherent HyperTransport device (such as *AMD-8131™ PCI-X® Controller*). These ports are mutually exclusive, and should be connected to only one other device. "*Istanbul*" also exposes two DRAM channel interfaces "DCT0" and "DCT1" to interface with system memory.

Contents of a BSD
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Configuration Options
See the following sections:

- Section 3.3, "Working with Device Groups", on page 18
- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Log Messages
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Difference from Real Hardware
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

7.29 AMD "Sao Paulo" Device

The AMD "*Sao Paulo*" device is a 8 core processor node, suitable for a G34 socket. It emulates a planned product that derives from a revision of the AMD Family10h product line. The device iteself is composed of 8 individual *AweSim Processor Devices* that are connected to a single "*AMD 8th Generation Integrated Northbridge Device*".

For more information on Group Devices, see Section 3.3, "*Device Groups*", on page 3.3.

Interface
AMD "*Sao Paulo*" has several connection ports. It has 4 HyperTransport links split to form 8 sub-links. Each sub-link can connect to a coherent HyperTransport device (such as another *AMD "Istanbul" Device*) or a non-Coherent HyperTransport device (such as *AMD-8131™ PCI-X® Controller*). These ports are mutually exclusive, and should be connected to only one other device. "*Sao Paulo*" also exposes two DRAM channel interfaces "DCT0" and "DCT1" to interface with system memory.

Contents of a BSD
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Configuration Options
See the following sections:

- Section 3.3, "Working with Device Groups", on page 18
- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Log Messages
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Difference from Real Hardware
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

7.30 AMD "Magny-Cours" Device

The AMD "*Magny-Cours*" device is a 12 core processor node, suitable for a G34 socket. It emulates a planned product that derives from a revision of the AMD Family10h product line. The device iteself is composed of 12 individual *AweSim Processor Devices* that are connected to dual *AMD 8th Generation Integrated Northbridge Devices*.

For more information on Group Devices, see Section 3.3, "*Device Groups*", on page 3.3.

Interface
AMD "*Magny-Cours*" has several connection ports. It has 4 HyperTransport links split to form 8 sub-links. Each sub-link can connect to a coherent HyperTransport device (such as another *AMD "Istanbul" Device*) or a non-Coherent HyperTransport device (such as *AMD-8131™ PCI-X® Controller*). These ports are mutually exclusive, and should be connected to only one other device. "*Magny-Cours*" also exposes four DRAM channel interfaces "DCT0", "DCT1", "DCT2" and "DCT3" to interface with system memory.

Contents of a BSD
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Configuration Options
See the following sections:

- Section 3.3, "Working with Device Groups", on page 18
- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Log Messages
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Difference from Real Hardware
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

7.31 AMD "DeerHound" Device

The AMD "*DeerHound*" device is a quad core processor node, suitable for an L1 socket. It emulates a production product that derives from a revision of the AMD Family10h product line. The device itself is composed of 4 individual *AweSim Processor Devices* that are connected to a single "*AMD 8th Generation Integrated Northbridge Device*".

For more information on Group Devices, see Section 3.3, "*Device Groups*", on page 3.3.

Interface
AMD "*DeerHound*" Device has several connection ports. It has 4 HyperTransport links split to form 8 sub-links. Each sub-link can connect to a coherent HyperTransport device (such as another *AMD "Istanbul" Device*) or a non-Coherent HyperTransport device (such as *AMD-8131™ PCI-X® Controller*). These ports are mutually exclusive, and should be connected to only one other device. "*DeerHound*" also exposes two DRAM channel interfaces "DCT0" and "DCT1" to interface with system memory.

Contents of a BSD
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Configuration Options
See the following sections:

- Section 3.3, "Working with Device Groups", on page 18
- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Log Messages
See the following sections:

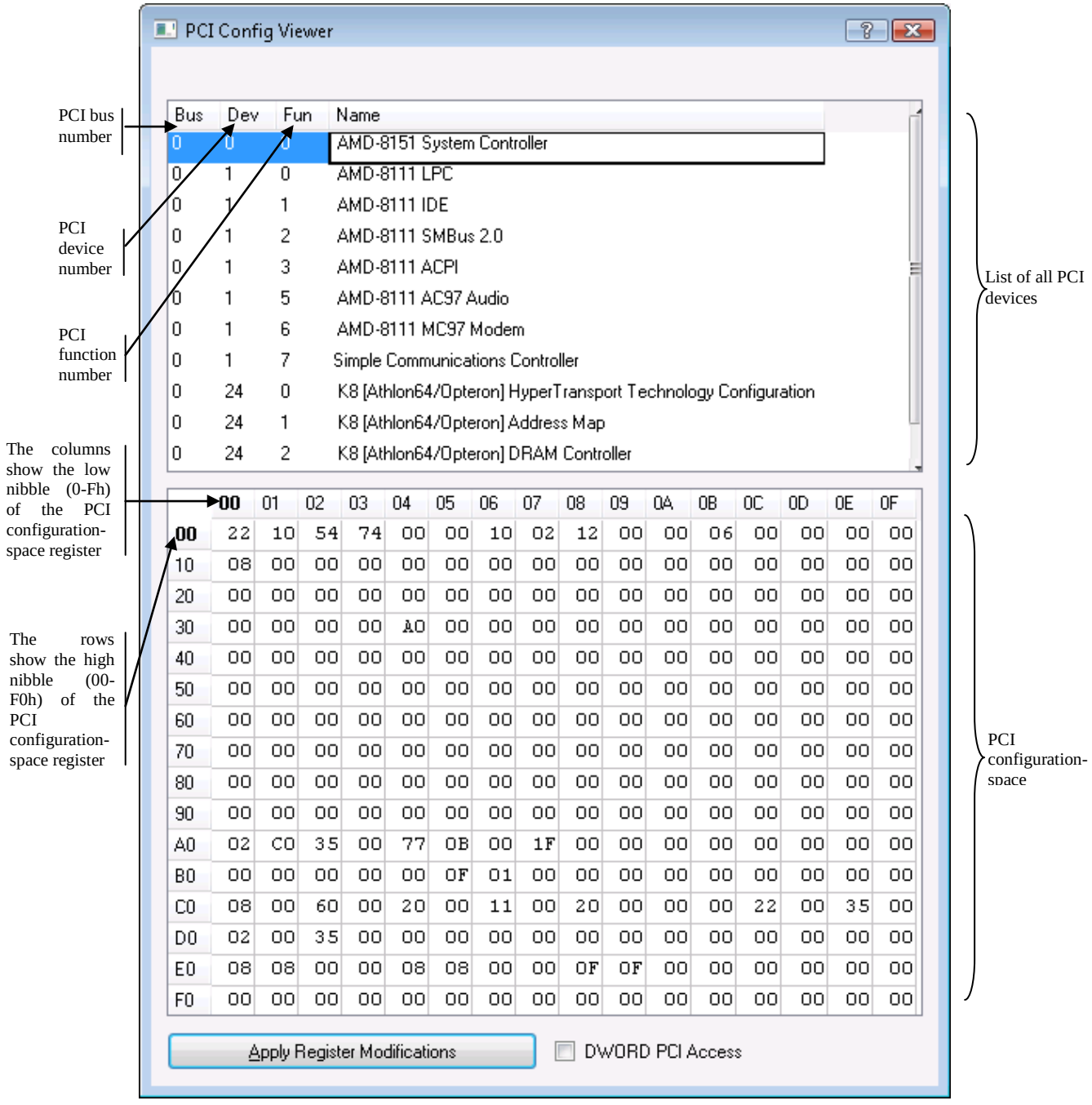
- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

Difference from Real Hardware
See the following sections:

- Section 7.1, "*AweSim Processor Device*", on page 55
- Section 7.10, "*AMD 8th Generation Integrated Northbridge Device*", on page 86

8 PCI Configuration Viewer

The *PCI Config Viewer* can be used to scan PCI buses and report information about the configuration-space settings for each PCI device.


PCI Config Viewer
PCI bus number
PCI device number
PCI function number
The columns show the low nibble (0-Fh) of the PCI configuration-space register
The rows show the high nibble (00-F0h) of the PCI configuration-space register
List of all PCI devices
PCI configuration-space
Apply Register Modifications
DWORD PCI Access

Bus	Dev	Fun	Name
0	0	0	AMD-8151 System Controller
0	1	0	AMD-8111 LPC
0	1	1	AMD-8111 IDE
0	1	2	AMD-8111 SMBus 2.0
0	1	3	AMD-8111 ACPI
0	1	5	AMD-8111 AC97 Audio
0	1	6	AMD-8111 MC97 Modem
0	1	7	Simple Communications Controller
0	24	0	K8 [Athlon64/Opteron] HyperTransport Technology Configuration
0	24	1	K8 [Athlon64/Opteron] Address Map
0	24	2	K8 [Athlon64/Opteron] DRAM Controller

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	22	10	54	74	00	00	10	02	12	00	00	06	00	00	00	00
10	08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	A0	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
A0	02	C0	35	00	77	0B	00	1F	00	00	00	00	00	00	00	00
B0	00	00	00	00	00	0F	01	00	00	00	00	00	00	00	00	00
C0	08	00	60	00	20	00	11	00	20	00	00	00	22	00	35	00
D0	02	00	35	00	00	00	00	00	00	00	00	00	00	00	00	00
E0	08	08	00	00	08	08	00	00	0F	0F	00	00	00	00	00	00
F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 8-1: PCI Configuration Viewer

8.1 Scanning PCI Buses

To view the *PCI Config Viewer* Dialog select the "*View→Show PCI Config Viewer*" entry from the *Main Window* menu. To scan a PCI bus, you must first load a bsd file that contains a CPU device. The dialog should look like the one shown in *Figure 8-1*.

8.2 Modifying the PCI Configuration contents

To modify the PCI configuration registers of a specific PCI device, select a device listed in the *PCI Config Viewers* list box. The *PCI Config Viewer* shows the contents of all PCI configuration registers of the selected device. To modify a certain byte of a PCI configuration register, click on the desired hex value and enter a new hex value. To apply the changes, click on the '*Apply Register Modifications'* button.

Read-only bits cannot be modified using the *PCI Config Viewer*. Modified values appear in red in the PCI configuration register list until you click on the '*Apply Register Modifications*' button or close the *PCI Config Viewer* dialog.

To change the byte view of the PCI configuration registers to a dword view, check the '*DWORD PCI Access*' check box.

9 Logging

The simulator provides support for three types of logging:

- A message log that can provide detailed text data from simulator devices and modules.
- An error log that provides text messages in response to critical errors or unexpected conditions.
- I/O Logging that provides detailed information about PCI Configuration, I/O and Memory Space accesses.

9.1 Message Log

The simulator shell provides an interface that loaded modules (devices and extensions) may use to report status and events. The messages may be displayed in a window, written to a file, or both. The information log may be enabled and disabled on a module-specific basis.

The informational log is controlled via the "*Message Log Window*" dialog box. To view this dialog, select the "*View→Message Log*" entry from the Main Window shell menu.

A sample of this dialog is shown in Figure 9-1:

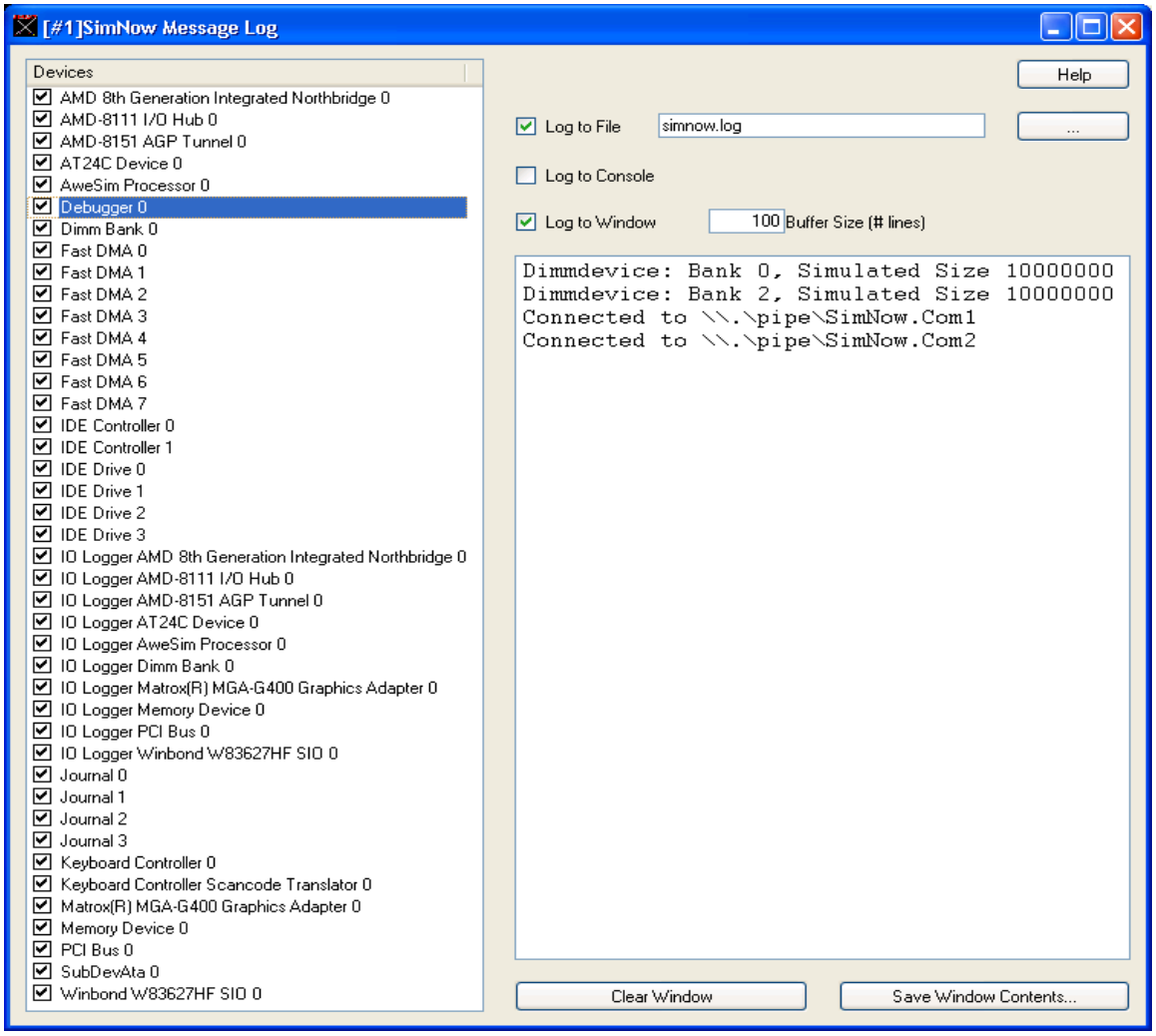


Figure 9-1: Message Log

The left-hand window lists all of the currently loaded modules. The user may individually enable or disable logging from a given module by using the checkbox next to the module's name. In addition, the user may configure module-specific logging options by double-clicking on the module name.

The top-right window contains three checkboxes which allow the user to control whether messages are displayed in the log window, written to a file, or logged to the AMD SimNow console. The bottom right window is used to display the informational message if the "*Log to Window*" option is selected.

To open the log file the first time a simulation is started, check the "*Log To File*" box is checked. The log file will remain open until one of the following events occurs:

- The BSD is closed or the simulator program terminates.
- The simulation is stared with the "*Log To File*" box unchecked.
- The simulation is started with a new log-file name specified.

9.2 Error Log

The simulator provides an interface that loaded modules may use to report critical errors or unexpected conditions. The messages are always written to a file, and the most-recent messages may be displayed in a window. The error log may not be disabled.

The most-recent error log entries may be viewed by selecting the "*View→Error Log*" entry from the Main Window menu, shown in Figure 9-2.

The error log file is enabled by checking the "*Log to File*" check box in the Message log dialog (Figure 9-2) and setting a filename for the error log. This file is created (or truncated to zero length if it already exists) and opened whenever a BSD file is opened or a new BSD is created. The error log is closed whenever the BSD is closed.

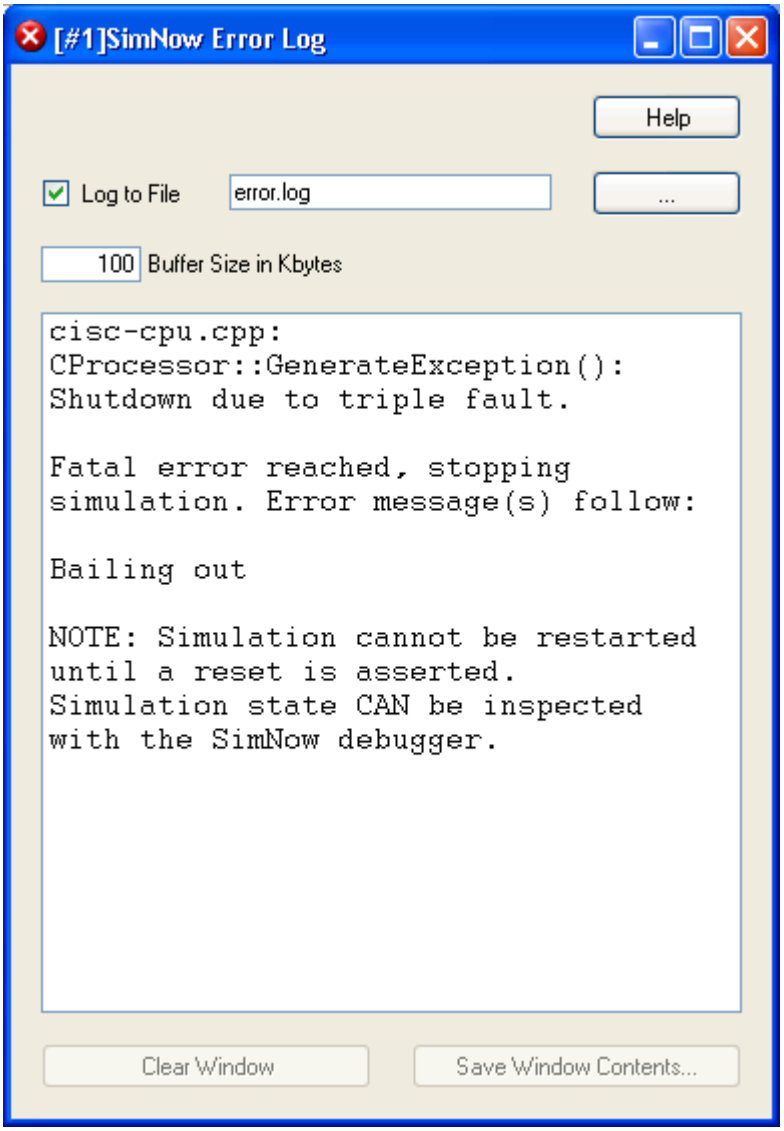


Figure 9-2: Error Log

9.3 I/O Logging

This is a generic feature available on all devices for logging slave accesses (i.e. accesses responded to by this device). Several categories of generic I/O logging are available. Logging is performed to the I/O loggers (see Section 9.1, "*Message Log*", on page 141) of names similar to the device you are enabling the logging for.

Caveat: Currently, devices which route to other devices may appear as if they are responding to the messages themselves, so bridge devices will likely log everything that is behind them.

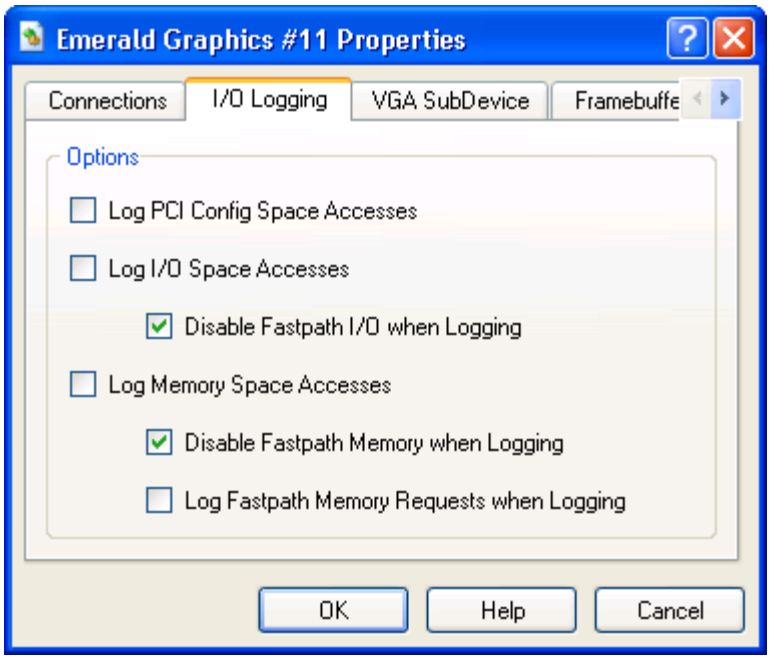


Figure 9-3: I/O Logging Dialog

Log PCI Config Space Accesses
Checking this will log PCI Config Space accesses made to the device.

Log I/O Space Accesses
Checking this will log I/O Space accesses made to the device. These are the accesses made with the x86 IO read/write instructions.

Disable Fastpath I/O when Logging
This item, checked by default, disables the Fastpath I/O mechanism when I/O Space Accesses logging is enabled. If this is unchecked, accesses may not appear in the log.

Log Memory Space Accesses
Checking this will log Memory Space accesses made to the device. These are the accesses corresponding to standard x86 *move*, *read* and *write* instructions to memory.

Disable Fastpath Memory when Logging

This item, checked by default, disables the Fastpath Memory mechanism when Memory Space Accesses logging is enabled. If this is unchecked, accesses may not appear in the log.

WARNING: Un-checking this item may lead to significantly compromised performance of SimNow if large numbers of accesses are being made to the device in question. For example, logging all accesses to the DIMM device would make SimNow extremely slow.

Log Fastpath Memory Requests when Logging
This item, when combined with un-checking *Disable Fastpath Memory when Logging*, will log both memory space accesses and Fastpath Memory requests themselves.

What is then logged are slow-path Memory Space Accesses and Fastpath Memory handle requests. Actual calls to Fastpath Memory, i.e. usage of Fastpath Memory handles, are not logged.

This page is intentionally blank.

10 CPU Debugger

10.1 Using the CPU Debugger

The CPU Debugger provides a list of commands and their descriptions when the "**?**" command is typed in the bottom line of the debug window, shown in Figure 10-1.

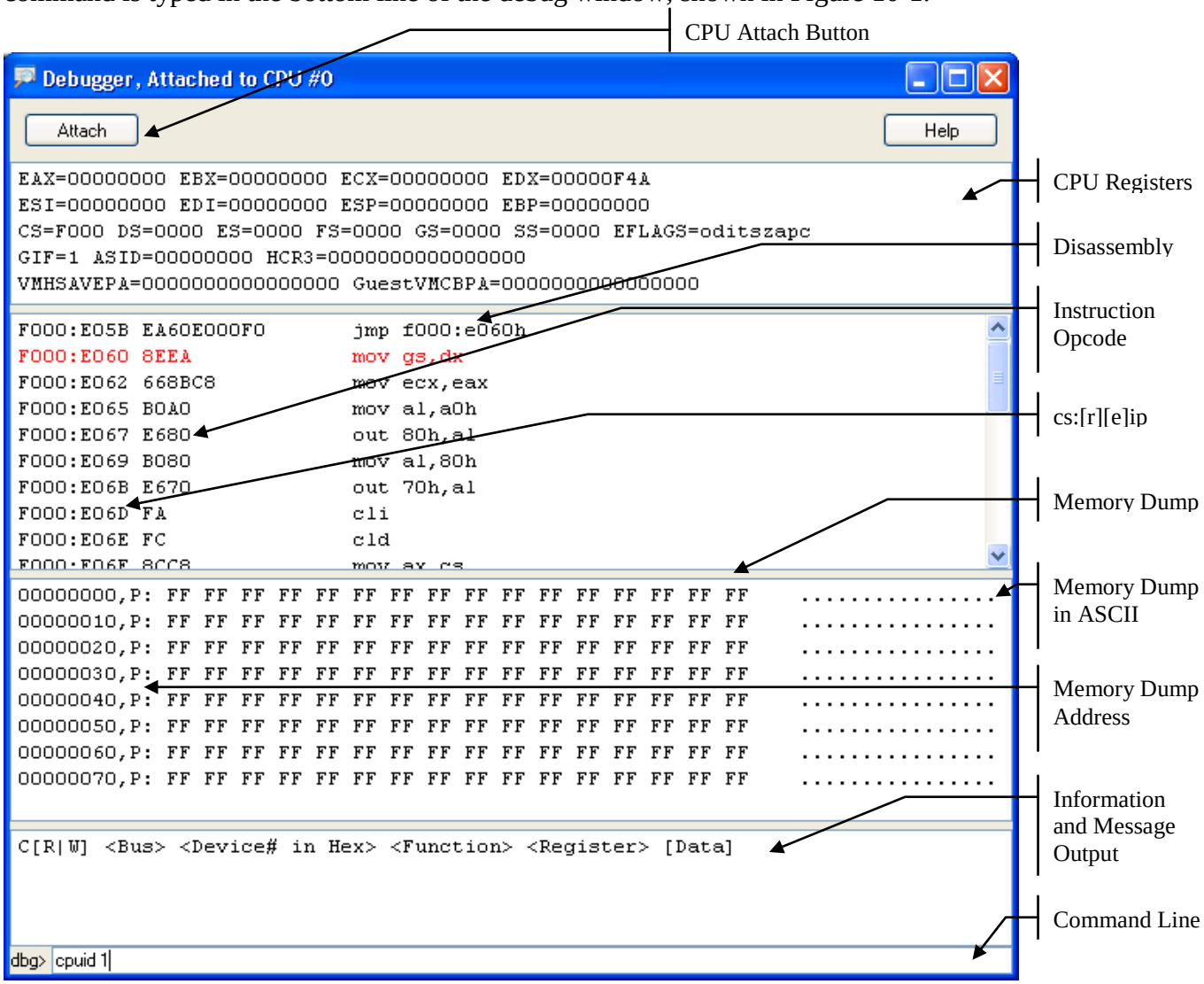

CPU Attach Button
Debugger, Attached to CPU #0
Attach
Help
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000F4A
ESI=00000000 EDI=00000000 ESP=00000000 EBP=00000000
CS=F000 DS=0000 ES=0000 FS=0000 GS=0000 SS=0000 EFLAGS=oditszapc
GIF=1 ASID=00000000 HCR3=0000000000000000
VMHSAVEPA=0000000000000000 GuestVMCBPA=0000000000000000
F000:E05B EA60E000F0 jmp f000:e060h
F000:E060 8EEA mov gs,dx
F000:E062 668BC8 mov ecx,eax
F000:E065 B0A0 mov al,a0h
F000:E067 E680 out 80h,al
F000:E069 B080 mov al,80h
F000:E06B E670 out 70h,al
F000:E06D FA cli
F000:E06E FC cld
00000000,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000010,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000020,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000030,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000040,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000050,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000060,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000070,P: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
C[R|W] <Bus> <Device# in Hex> <Function> <Register> [Data]
dbg> cpuid 1
CPU Registers
Disassembly
Instruction Opcode
cs:[r][e]ip
Memory Dump
Memory Dump in ASCII
Memory Dump Address
Information and Message Output
Command Line

Figure 10-1: CPU Debugger Window

10.1.1 Setting a Breakpoint

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Open the Debugger Window ("*View→Show Debugger*") or click on . The simulation will pause and the Debugger Window will appear.
3. The bottom pane in the CPU Debugger Window is the debugger command line. Enter a *BX*, *BM*, or *BI* on the debugger command line to setup and enable a breakpoint. The *BX*, *BM*, and *BI* commands specify breakpoints on execution, data access, or I/O access, respectively. Each of these commands requires an

address parameter that specifies a linear address associated with the breakpoint. An optional parameter can be used to specify the pass count, i.e., the number of times the breakpoint should be hit before breaking into the debugger. In addition, the *BM* and *BI* commands accept an optional parameter that specifies whether to break on a read/input, or write/output transaction to the specified address. Examples of each command are shown in Table 10-1.

4. After setting up and enabling the breakpoint(s), enter *G* on the command line to resume CPU execution. This will execute the debugger's Go command, returning the CPU to continuous execution. If a breakpoint is hit, the simulation will pause, and the debugger will gain attention.

Command	Description
`BX 1234abcd`	Break on the next execution of the instruction located at linear address, 0x1234ABCD.
`BX 1234ABCD 2`	Break on the third execution of the instruction located at linear address, 0x1234ABCD.
`BM abcd1234 r 3`	Break on the fourth read of the memory location, 0xABCD1234 (linear).
`BM abcd1234 3`	Break on the fourth access (read or write) of the memory location, 0xABCD1234 (linear).
`BI 80 w 3`	Break on the fourth write to I/O address, 0x80.

Table 10-1: Debugger Breakpoint Command Examples

10.1.2 Single Stepping the Simulation

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Open the Debugger Window ("*View→Show Debugger*") or click on . The simulation will pause and the Debugger Window will appear. .
3. The bottom pane in the Debugger Window is the debugger command line. When the Debugger Window has attention, enter T on the debugger command line. The debugger Trace command will execute, causing the CPU device to execute one instruction, and then return attention to the debugger.
4. The debugger will repeat the last entered command, if you just type Enter on the command line. So, you can repeatedly step instructions by entering *T* once, then repeatedly hitting the Enter key.
5. The simulation can be returned to continuous execution by entering *G*). This executes the debugger's *Go* command.

10.1.3 Stepping Over an Instruction

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Open the Debugger Window ("*View→Show Debugger*") or click on . The simulation will pause and the Debugger Window will appear.
3. When the Debugger Window has attention, enter *P* on the debugger command line. The debugger *Pretty Trace* command will execute, causing the CPU device to execute up to the next instruction in linear order (i.e., step over calls, interrupts, repeated instructions, and loops). This is distinguished from the *T* command,

which will step into calls, interrupts, etc., executing the next instruction regardless of its type.

4. The debugger will repeat the last entered command, if you just type Enter in the command edit window. So, you can repeatedly execute the pretty trace command by entering P once, then repeatedly hitting the Enter key.
5. The simulation can be returned to continuous execution by entering G. This executes the debugger's *Go* command.

10.1.4 Skipping an Instruction

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Setup a breakpoint to break at the instruction that you want to step over (see Section 10.1.1, "*Setting a Breakpoint*", on page 147). Execute to the breakpoint.
3. Determine the EIP of the next instruction after the one to be skipped. This can easily be determined by viewing the disassembly listing in the debugger. The top line in the disassembly listing is the instruction pointed to by the current EIP (the instruction that you wish to skip).
4. Use the debugger's *R* command to change the value in the EIP register. This can be done by typing *R EIP = new_value* on the debugger command line. In this case, *new_value* is the linear address of the instruction that follows the one that you want to skip.
5. Enter *G* on the debugger command line. This will execute the debugger's *Go* command. CPU execution will resume.

10.1.5 Viewing a Memory Region

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Open the Debugger Window ("*View→Show Debugger*") or click on . The simulation will pause and the Debugger Window will appear.
3. When the Debugger Window has attention, use the debugger's *DB*, *DW*, *DD*, or *DQ* command to display the contents of a memory region in the debugger. The second letter of the command specifies the display format for the dump. The *DB* command displays byte format, *DW* displays word format, *DD* displays dword format, and *DQ* displays qword format. Each of these commands requires a second parameter that specifies the beginning address (in hex) of the memory dump. A linear address can be specified by adding a ',L' suffix to the address. Similarly, a physical address can be specified by adding a ',P' suffix to the address. Examples of the memory-dump commands are shown in Table 10-2.
4. After the first memory range is displayed, you can repeatedly hit *Enter* to advance the display to the next sequential memory block.

Command	Description
`DB 010,p`	Dump memory in byte format, starting at physical address, 0x00000010.
`DW abcd1234,L`	Dump memory in word format, starting at linear address, 0xABCD1234.
`DQ c001c0de,L`	Dump memory in quad word format, starting at linear address, 0xC001C0DE.

Table 10-2: Debugger Memory Dump Command Examples

When using Pacifica Virtualization Technology in simulation, the user can tell the debugger to access memory for either the guest or the host. If multiple guests are running under a hypervisor, the debugger will acess memory for the last guest that has run. The user can further qualify an input address using the 'G' (Guest) and 'H' (Host) specifiers. For example:

Command	Description
`Dd c001c0de,HL`	Dump the SVM host linear memory starting at address 0xC001C0DE.
`Dd c001c0de,GL`	Dump the last SVM guest linear memory starting at address 0xC001C0DE.
`Dd c001c0de,HP`	Dump the SVM host physical memory starting at address 0xC001C0DE.
`Dd c001c0de,GP`	Dump the last SVM guest physical memory starting at address 0xC001C0DE.

Table 10-3: Debugger Pacifica Memory Dump Command Examples

If the user omits the 'G' or the 'H' specifier, the debugger will access memory from the perspective of the attatched CPU's current state.

10.1.6 Reading PCI Configuration Registers

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Open the Debugger Window ("*View→Show Debugger*") or click on . The simulation will pause and the Debugger Window will appear.
3. Use the debugger's *S* command to view the PCI configuration register contents for a particular PCI function. The *S* command takes three hex parameters: bus, device, function. If the specified bus, device, and function exist in the simulated system, the debugger will display all 256 bytes of configuration data.

10.1.7 Reading CPU MSR Contents

1. Stop the simulation as described in Section 3.1, "*Tool Bar Buttons*", on page 7.
2. Open the Debugger Window ("*View→Show Debugger*") or click on . The simulation will pause and the Debugger Window will appear.
3. Use the debugger's *R* command to view the contents of an MSR. This can be accomplished by typing *R* M*address* on the debugger command line. In this case, *address* is the 32-bit address (in hex) of the MSR. All leading zeros must be typed in the address. Examples of MSR reads are shown in Table 10-4:

Command	Description
`R M00000250`	Displays the contents of the MSR with an address of 0x0250.
`R MC001001A`	Displays the contents of the MSR with an address of 0xC001001A.

Table 10-4: MSR Read Examples

4. MSR registers can be modified by adding a "= *Value*" suffix on the above command syntax. *Value* will be assigned to the MSR register only if the value does not modify any reserved bits in the MSR. If an attempt is made to modify any reserved bits, the MSR write is ignored. An example MSR write is shown in Table 10-5:
5. This command may not allow access to all MSRs that are supported by the CPU model. To view a list of all registers supported by the *R* command, enter *R?* on the debugger command line.

Command	Description
`R MC001001A = 0000000004000000`	Assigns a value of 0x0000000004000000 to the MSR with an address of 0xC001001A.

Table 10-5: MSR Write Example

10.1.8 Find Pattern in Memory

The find pattern command **q1** and **qa** can be used to search for a specific pattern in memory, The pattern that is searched for can either be an ASCII string or a binary pattern. If the search is for an ASCII string the *noncase* option (see Table 10-7, "*Debugger Commands and Definitions*", on page 154) can be used to match any character.

Command	Description
`q1 0x1000,L 0x2000 "PCI"`	Finds the first occurrence of ASCII pattern "PCI" in the given memory range, 0x1000 - 0x2000.
`qa noncase 0x1000,L 0x2000 "PCI"`	Same as above but finds all occurrence of the ASCII pattern "PCI" using the none case-sensitive search algorithm.
`qa 0xF0000,P 0xFFFF 0x55 0xAA`	Finds all occurrences of the binary-pattern 0x55 0xAA in the given memory range, starting at physical address 0xF0000 and ends at 0xF0000+0xFFFF.

Table 10-6: Find Pattern Example

10.2 Debugger Command Reference

The CPU Debugger Window consists of five areas, as shown in Figure 10-1. The top-most area displays the current CPU integer registers in 16-, 32-, or 64-bit mode, depending on the current mode of the CPU. The next area displays a disassembly of the next six instructions, starting at the current CS:[R|E]IP. The next area displays 128 bytes of memory, as bytes, words, dwords, or qwords. The address, size, and physical or virtual attributes are based on the most recent *D* command. The next area is a general message window where messages and information are displayed. The bottom area is the command area, where debugger commands are entered.

Table 10-7 lists the debugger commands and their definitions.

Debugger Command	Definition
?	Displays an abbreviated list of the available commands and their syntax.
<blank line>	Repeat of previous command.
*<automation command>	Execute an automation command.
#P <Path> [;<Path>	Sets the file search path.
#L <Symbol File> [Load Address]	Loads the named symbol file, optionally offsetting each address by the given load offset. When the load is completed, the module name attached to this group of symbols is displayed. Supported symbol file extentsions are "*.TXT", "*.SYMTEXT", and Linux "symbol.map" file ("*.MAP").
#M	Displays a list of the symbol modules currently loaded.
#U <Module Name>	Unloads the named symbol module that had previously been loaded with the #l command.
#? <Symbol>	Displays all symbols that contain the given string.
#! <Address>	Displays the symbol that most closes matches the given address.
bc {* | list }	Clears one or all breakpoints.
bd {* | list }	Disables one or all breakpoints.
be {* | list }	Enables one or all breakpoints.
bf <vector> <Pass count>	Creates and enables a breakpoint for the indicated CPU exception. Sets the pass count to [count], or 0 if not specified.
bh <vector> <Pass count>	Creates and enables a breakpoint for the indicated hardware interrupt. Sets the pass count to [count], or 0 if not specified.
bi <address> [r | w] <Pass count> [v[b|w|d] <data>]	Creates and enables a breakpoint for the indicated I/O address. Sets the pass count to [count], or 0 if not specified. Defaults to read/write, but can be set to read-only or write-only using the [r] or [w] options. [v] enables the data <data> check capability for [b]yte, [w]ord, or [d]ouble word I/O accesses. For example, “bi 80 w vb c0” stands for break when byte 0xC0 is written to I/O port 0x80.
bl [* | list]	Display the settings of one or all breakpoints.

Debugger Command	Definition
`bm <address> [r | w] <Pass count> [v[b|w|d] <data>]`	Creates and enables a breakpoint for the indicated memory address. Sets the pass count to [count], or 0 if not specified. Defaults to read or write, but can be set to read-only or write-only using the [r] or [w] options. [v] enables the data <data> check capability for [b]yte, [w]ord, or [d]ouble word memory accesses. For example, "bm 1000 w vb c0" stands for break when byte 0xC0 is written to memory address 0x1000.
`bs <Vector> <Pass count>`	Creates and enables a breakpoint for the indicated software interrupt vector. Sets the pass count to [count], or 0 if not specified.
`bx <address> <Pass count>`	Creates and enables a breakpoint for the indicated code fetch address. Sets the pass count to [count], or 0 if not specified. Sets the pass count to [count], or 0 if not specified.
`c[r|w] <Bus> <Dev> <Func> <Off> [data]`	Performs a PCI configuration [r]ead or [w]rite.
`d[b|w|d|q] <address range>[,[l|p]`	Displays the contents of [p]hysical (default) or [l]inear memory as [b]ytes, [w]ords, [d]ouble words, or [q]uad words, or in the previous format if not specified.
`e[b|w|d|q] <address> <data ...>[,[l|p]`	Allows the modification of [p]hysical (default) or [l]inear memory, in [b]ytes, [w]ords, [d]ouble words, or [q]uad words, or in the previous format, if not specified. Data values are entered immediately after the address, separated by spaces.
`f[b|w|d|q] <address range> <value> [,[l|p]`	Fills the given [p]hysical (default) or [l]inear memory-range with the indicated *value*.
`g [address]`	Begins or will resume CPU execution, setting a temporary execution breakpoint on the given *address*.
`h [on | off | clear | <value>]`	Controls history-trace collection. [ON] enables trace collection and clears the current trace buffer; [OFF] disables trace collection, and [CLEAR] clears the current trace buffer. Specifying no arguments, or a value, disassembles the most recent <value> instructions executed.
`i[b|w|d] <port>`	Input a [b]yte, [w]ord, or [d]ouble word from the indicated port.
`o[b|w|d] <port> <data>`	Output a [b]yte, [w]ord, or [d]ouble word to the indicated port.

Debugger Command	Definition
`p`	Similar to the *t* command, single steps the simulation one instruction, unless the current instruction is a call, software interrupt, or repeated string instruction, in which case this command sets a temporary execution breakpoint at the instruction sequentially following the current instruction, and starts simulation.
`r [regname[= <value>]]`	Displays, and optionally alters, the contents of various CPU registers. For a list of register names that are supported, type *R?*. Normally, the display is in the current CPU mode. To force 16-bit, 32-bit, or 64-bit register display, type *R16, R32*, or *R64* respectively.
`R16`	Display 16-bit registers (until the next instruction).
`R32`	Display 32-bit registers (until the next instruction).
`R64`	Display 64-bit registers (until the next instruction).
`s <Bus> <Device> <Function>`	Displays the PCI configuration registers associated with the given Bus, Device, and Function number.
`t [count]`	Executes [count] instructions. The default value for [count] is 1.
`u [address range]`	Disassembles instructions starting, at the given address and continuing for [length] instructions. Instructions are disassembled using the current CPU execution mode.
`v`	Displays the version number information for the attached processor device.
`q<a|1> [noncase] <StartAddress>[,[l|p]] <[[L]Length] | [EndAddress]> <Pattern>`	Search physical (default) or linear Memory for pattern and display all or only first occurrence(s).

Table 10-7: Debugger Commands and Definitions

In general, address and count values can be specified as constants (hex for addresses, ports, and values; decimal for counts and lengths), or as register names. For addresses, the CS:, DS:, ES:, FS:, GS:, and SS: prefixes are also allowed.

Address values may be suffixed by ‘,L’ to specify a linear address (the default) or ‘,P’ to specify a physical address. Addresses may also be specified by their symbol name. Precede the symbol name with a # character to distinguish it from a hex constant.

11 Debug Interface

The simualtor supports Linux and Windows® based debugging. It is recommended to use the GDB interface to debug on Linux based hosts. The kernel debugger interface can be used to debug on Windows based hosts.

11.1 Kernel Debugger

This only applies to the Windows® version of the simulator and **not** to the Linux version.

The simulator can interact with the kernel debugger through:

- EXDI interface (see Section 7.20, "*EXDI Server Device*", on page 108).
- Serial port connection.

The serial ports can be configured so that any data read from or written to the simulated serial ports is made available to the host machine. The serial ports can each be configured to do this using either a named-pipe, or the actual serial port hardware.

The automation commands "GetCommPort" and "SetCommPort" are used for this purpose, see Section A.7.10, “Serial”, on page 232.

Use the serial ports "SetCommPort" command to set the simulated serial port to use a specific COM port. For example, to set the second serial port in the simulation to use COM4 for its communication, you would type

```
Serial:1.SetCommPort COM4 57600
```

The simulator will program the appropriate COM port (COM4 in the above example) to 57600 baud, 8 bits, no parity, 1 stop bit, no flow control.

All characters transmitted by the simulation through the serial port (second serial port in the above example) will be sent out to the given COM port (COM4 in the above example). In the same manner, all data received by the simulator through the given COM port (COM4 in the above example) will appear as received data in the simulated COM port.

To set the simulated serial port (COM1) to use a named-pipe you would type

```
Serial:1.SetCommPort pipe
```

The simulator will program the appropriate COM port (COM1 in the above example) to use the named-pipe “\\.*pipe**SimNow.Com1*” on the host to transfer data between host and the simulated machine.

The pipe is not created until the first “go” command will be executed. This can be achieved by clicking on the “go” button followed by a click on the “stop” button. This command sequence will setup the named-pipe.

If you try to connect the kernel debugger without setting up the named-pipe as described the kernel debugger will return an error message.

In case you have difficulties to establish a connection, or the connection is unstable, or KD has difficulties to stay in sync with the simulated OS. You can set a *multiplier* to delay the baud rate. The baud rate is normally modeled based on the time elapsed on the simulated system. The simulated system may be running at 1/100 of normal time which will give longer time delays than the kernel debugger can tolerate. Consequently we provide a way to speed up the modeled baud rate by up to 100 times. For example to delay the baud rate by 1/100th of normal you would use the following automation command:

```
Serial:1.SetMultiplier 1
```

By default, the *multiplier* is 100 which means the modeled rate is unchanged. The user may set it in the range 1 to 100. When set to 1, the modeled rate is 100 times faster than the baud rate, so the system delays will be that much shorter. See also Section A.7.10, “*Serial*”, on page 232.

The following command will connect the kernel debugger to the simulator using a pipe as communication channel:

```
C:\Program Files\Debugging Tools for Windows 64-bit\kd -k
com:pipe,port=\\.\pipe\SimNow.Com1
```

We recommend not starting the kernel debugger too early. To achieve best results launch the kernel debugger after the O/S kernel has loaded and it is trying to establish a connection with the kernel debugger.

11.2 GDB Interface

Getting the gdb interface in the simulator to work involves a sequence of commands in both the simulator and gdb. The current implementation requires the simulator to be started and told to be ready for gdb to connect and then having gdb connect. As long as the gdb command, "*target remote* ..." is issued last, the interface should be established.

It has been observed that after shutting down the simulator, the port used by the gdb interface may not become immediately available for reuse. If this happens just shut down both the simualtor and gdb and start again and the problem should go away.

11.2.1 Simple Approach

This assumes you are running the simulator and gdb on the same machine.

- Start the simualtor

- Run the following automation command:
```
1 simnow> shell.gdb <ENTER>
```

- Start gdb
```
gdb> set architecture i386:x86-64 <ENTER>
gdb> target remote:2222 <ENTER>
```

11.2.2 Alternate Approach

This assumes you are running the simualtor and gdb on the same machine.

- Start the simulator
- Run the following automation command:
```
1 simnow> shell.gdb <ENTER>
```

- Add the following to your *.gdbinit file*
```
define simnow
     set architecture i386:x86-64
     target remote:2222
end
```

- Start gdb
```
gdb> simnow <ENTER>
```

11.2.3 Using Another Port on the Same Machine

The simualtor defaults to using port 2222 but can be directed to use another port.

- Start the simulator
- Run the following automation command:
```
1 simnow> shell.gdb 2233 <ENTER>
```

- Start gdb
```
gdb> set architecture i386:x86-64 <ENTER>
gdb> target remote:2233 <ENTER>
```

11.2.4 Using Two Separate Machines

- Start the simualtor on *simnow-host*
- Run the following automation command:
```
1 simnow> shell.gdb <ENTER>
```

- Start gdb on *gdb-host*
```
gdb> set architecture i386:x86-64 <ENTER>
gdb> target remote simnow-host:2222 <ENTER>
```

11.3 Linux Host Serial Port Communication

When running the simulator on a Linux host, the serial port is able to communicate with external host applications via either a named-pipe or the host serial port. If the user has configuired named-pipe communication, the simualtor will set up an input pipe and an

output pipe at "`~./simnow/comX/simnow_in`" and "`~./simnow/comX/simnow_out`". External applications should read data from the simulation using the *simnow_out* named-pipe. Conversely, external applications should send serial data to the simulation using the simnow_in pipe.

Note that it is not possible for two simualtor sessions to communicate with each other on the same host using named-pipes. This is an issue that will be fixed in a future version of the simulator.

When the simaultor serial port has been configuired to use the host serial port, the simualtor will open "`/dev/ttyS0`" or "`/dev/ttyS1`" (depending on wether it is COM1 or COM2). Note that the user will need to be running the simulator with root privelages to avoid an access denied error when the simualtor attempts to open the device. The simulator can communicate with external applications, such as a kernel debugger in this mode.

12 Command API

The *CMDAPI* (cmdapi.dll) gives Windows users a way to script the simulator using any scripting language that can interface with the Microsoft Component Object Model (COM). It gives you the opportunity to create scripts that instantiate a simulator object. You can use this instantiated object to execute any of the SimNow™ automation commands, see Section A.7, "*Automation Commands*", on page 224.

CMDAPI is installed and registered whenever a SimNow release package has been installed successfully.

After instantiating a *SimNow.Command* object, you can use the following methods to execute automation commands, and retrieve status.

Exec

The *Exec* method executes the automation command that *arg1* contains.

```
bool Exec(arg1, arg2);
```

Parameters

arg1

A string that contains the SimNow automation command to execute. For example, "debug:0.execcmd t".

arg2

An input string buffer in which SimNow is to place the response from the command in *arg1*.

Return Value

Returns *true* if command completed successfully; otherwise it returns *false*.

GetLastError

The *GetLastError* method returns the last error code. If *Exec* returns *false* you can call *GetLastError* to retrieve the error code.

```
void GetLastError(arg1);
```

Parameters

arg1

An input string buffer, in which SimNow will place the last error that was recorded from the automation interface.

The Perl code in Example 12-1 shows how to instantiate a *SimNow.Command* object and how to interact with the SimNow™ CMDAPI interface.

```
#!perl -w
```

```
use Win32::OLE;
use Win32::OLE::Variant;

$Win32::OLE::Warn = 3;

$cmd = Win32::OLE->new('SimNow.Command')
       or die "Cannot open SimNow.Command\n";

$MyResponse = Variant(VT_BSTR | VT_BYREF, "");

do {
      print "simnow> ";
      $CmdLine = <>;
      chomp($CmdLine);
      if ($CmdLine)
      {
            if ($cmd->Exec($CmdLine, $MyResponse))
            {
                  print "$MyResponse\n";
            }
            else
            {
                  $cmd->GetLastError($MyResponse);
                  print "Cannot Exec: $MyResponse\n";
            }
      }
} while ($CmdLine);

print "\ndone\n";
```

Example 12-1: Perl Sample CMDAPI Source Code

13 DiskTool

Use the DiskTool utility to create hard-disk images. DiskTool copies, byte-for-byte, the contents of a secondary hard disk into an *.hdd* file. This *.hdd* file can be loaded as a disk image in the simulator.

DiskTool runs in two modes, GUI mode, and command-line mode. Double-clicking on the DiskTool icon, or running DiskTool from the command line with no command line options, starts DiskTool in GUI mode. If you run DiskTool from the command line and include any command-line parameters, DiskTool runs in command line mode. To get a list of the command-line options, run "*DiskTool -help*".

13.1 Command-Line Mode

The functions recognized by the DiskTool command line include:

Option:
G = Copy a physical device to the given image file.

Syntax:
{/G|-G} <DeviceName> <ImageName> [ImageSize]

[ImageSize] = # of sectors of data to copy from the device to the image file
0 = All sectors (this is the default value)
1 = All data to the end of physical partition 1
2 = All data to the end of physical partition 2
3 = All data to the end of physical partition 3
4 = All data to the end of physical partition 4
<Any Other Valid Number> = The number of sectors specified

Example:
disktool –g /dev/hd0 image.hdd 102400

This command reads the first 102400 sectors from device */dev/hd0* and places them in the image file, *image.hdd*.

Option:
P = Put the image file <ImageName> to physical device <DeviceName>.

Syntax:
{/P|-P} <DeviceName> <ImageName>

Example:
disktool –p /dev/hd0 image.hdd

This command reads image file *image.hdd* and writes data to physical device */dev/hd0*.

Option:
E = Erase (Write zeros to all blocks) physical device.

Syntax:
{/E|-E} <DeviceName>

Example:
disktool –e /dev/hd0

This command writes zeros to all sectors on device */dev/hd0*.

Option:
N = Create a new blank image file that represents a freshly formatted device.

Syntax:
{/N|-N} <ImageName> <ImageSize>

Example:
disktool –n image.hdd 102400

This command creates an image file named *image.hdd* that represents a physical hard-disk drive containing 102400 sectors (each sector is 512 bytes).

13.2 GUI Mode

The DiskTool GUI window is shown in Figure 13-2. DiskTool will only display floppy drives, and DVD/CD and HDD drives that are connected to either the primary or the secondary IDE controller. It will not display external USB or firewire drives, drives attached to SCSI controllers, etc.

DiskTool displays the names of these devices in the *Physical Drives* list box, using names appropriate for the host operating system. When running under Windows, the *Physical Drives* list box will show you the physical drives, and in parenthesis, the logical drive letters that are associated with the partitions on that drive. Selecting any of these physical devices causes DiskTool to display information about that device in the lower *Drive Information* list box.

DiskTool also displays information about all identified devices in a shell window. The DiskTool shell window is shown in Figure 13-1.

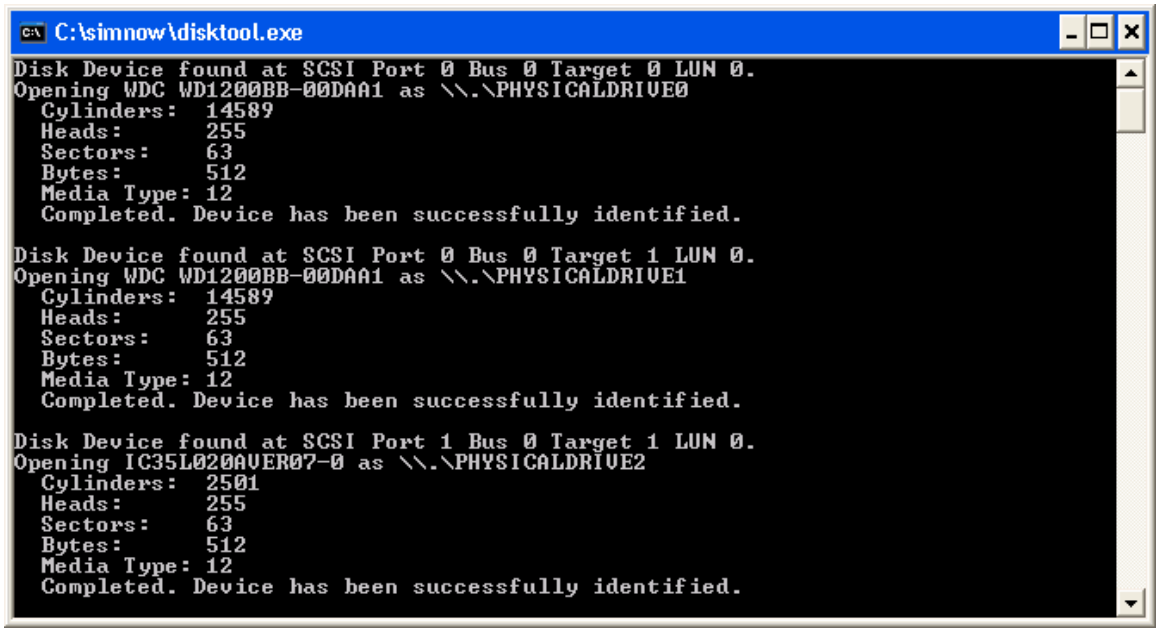

C:\simnow\disktool.exe
Disk Device found at SCSI Port 0 Bus 0 Target 0 LUN 0.
Opening WDC WD1200BB-00DAA1 as \\.\PHYSICALDRIVE0
Cylinders: 14589
Heads: 255
Sectors: 63
Bytes: 512
Media Type: 12
Completed. Device has been successfully identified.
Disk Device found at SCSI Port 0 Bus 0 Target 1 LUN 0.
Opening WDC WD1200BB-00DAA1 as \\.\PHYSICALDRIVE1
Cylinders: 14589
Heads: 255
Sectors: 63
Bytes: 512
Media Type: 12
Completed. Device has been successfully identified.
Disk Device found at SCSI Port 1 Bus 0 Target 1 LUN 0.
Opening IC35L020AVER07-0 as \\.\PHYSICALDRIVE2
Cylinders: 2501
Heads: 255
Sectors: 63
Bytes: 512
Media Type: 12
Completed. Device has been successfully identified.

Figure 13-1: DiskTool Shell Window

DiskTool will only copy drives - not partitions, although it does have the ability to stop copying at the end of a given partition. So, for example, you can copy the contents of a drive starting at the beginning of the drive and ending at the end of the 2nd partition, but you can not copy only the 2nd partition.

LINUX Note: The list box always shows */dev/fd0* and */dev/fd1*. If you click on one of these and the physical device does not actually exist, the GUI will hang for a short time, and will then display information in the lower list box indicating that a 4Kb media is installed in this device DiskTool only recognizes device names */dev/hda* through */dev/hdz*. In addition, it looks for the file */proc/ide/hd?/media*, and uses the information in that file to determine whether the device is a hard drive or a DVD/CD drive. If the file does not exist, or if its contents cannot be parsed, the device will not be listed.

The buttons on the right side of the DiskTool Window correspond to the four command line options listed above. In addition, there are *About* and *Exit* buttons that perform the obvious function.

When creating a new blank image, or when getting an image from a physical device to an image file, an additional dialog is presented that allows you to select how large the new image file should be. The options in this dialog mirrors the [*Image Size*] options for the equivalent command line-commands.

After launching DiskTool, you are presented with the interface, shown in Figure 13-2.

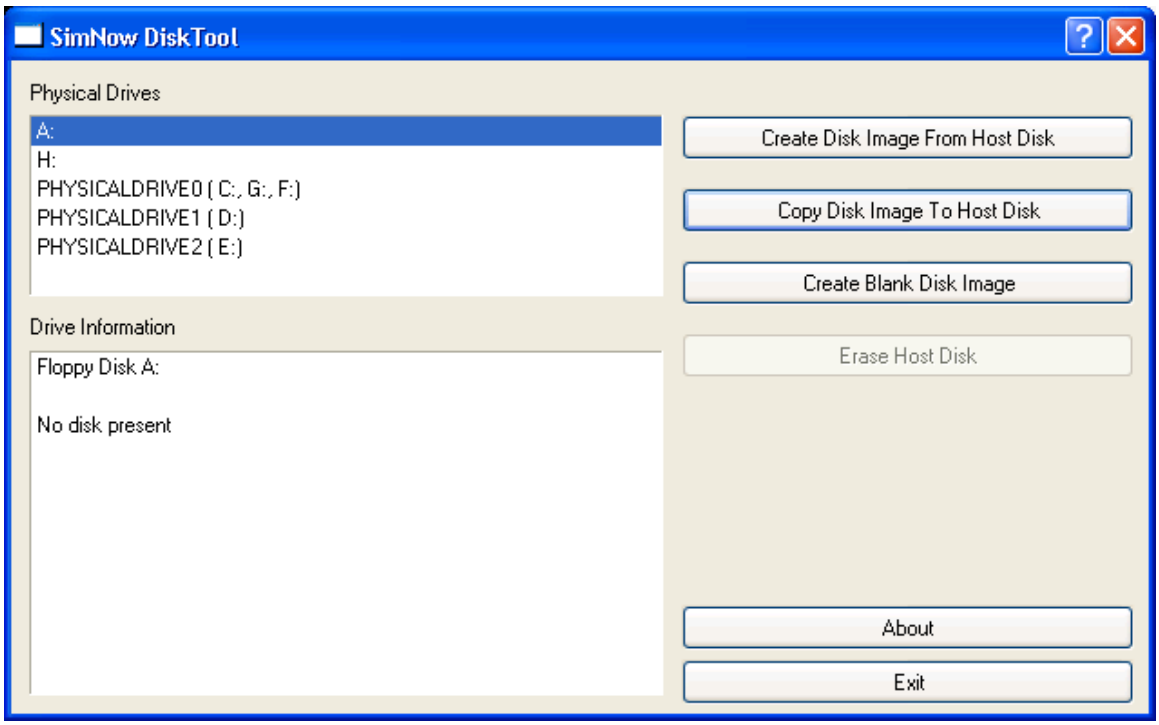


Figure 13-2: DiskTool GUI Window

You may select any physical drive in your system, including floppy drives. Selecting a drive updates the *Drive Information* list box as shown in Figure 13-3.

Note: DiskTool does **not** support Serial ATA (SATA) drives!

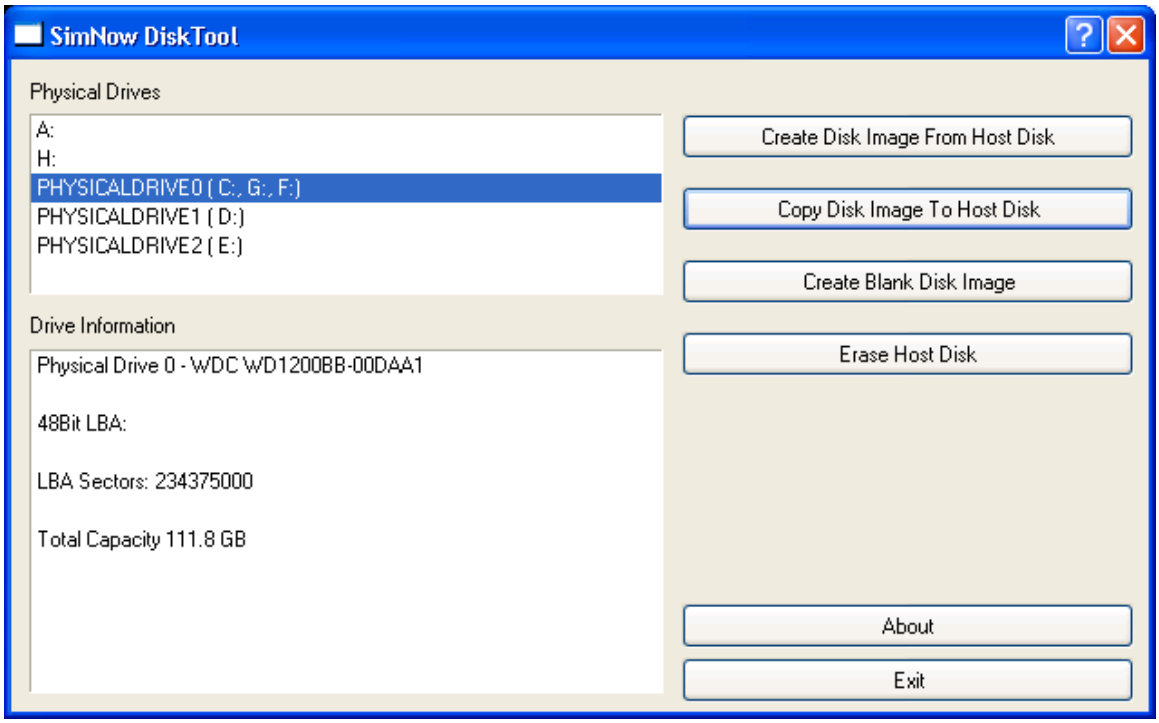


Figure 13-3: DiskTool Drive Information

When a drive is selected, you have the option to get an image from the drive, put an image onto the drive, or erase the contents of the drive.

If you erase the contents of the drive, a dialog will ask for confirmation that you actually wish to permanently destroy the contents of that hard disk.

In case DiskTool displays an "Operation failed!" message box, DiskTool was unable to lock or unlock the drive. This can happen if, for example, any files or explorer windows are open on any of the partitions on the selected drive.

For example, if the drive that DiskTool is trying to access has partitions for C: and D:, and an explorer window is open on any path within D:, then DiskTool won't be able to lock or unlock that drive, and DiskTool will display an "Operation failed!" message box.

If you put an image onto the drive, a dialog will again ask for confirmation that you actually wish to permanently destroy the contents of that hard disk. Then a dialog prompts for the location of the image file that should be placed on that hard disk. A progress bar (Figure 13-4) will inform you of the progress being made.

If you get an image from a drive, a dialog window will prompt for the path of file that will store the disk image. A progress bar will inform you of the progress being made.

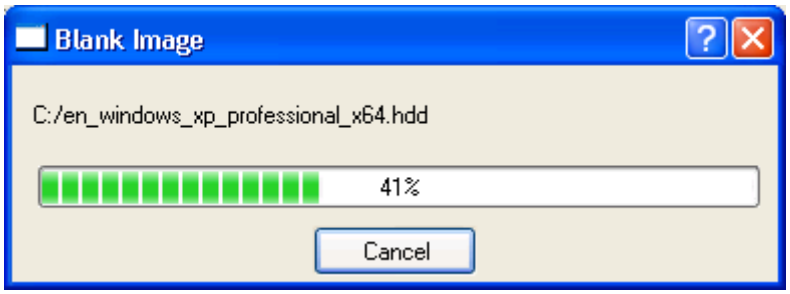


Figure 13-4: DiskTool Progress Window

This page is intentionally blank.

14 BIOS Developer's Quick Start Guide

This section provides you with instructions on how to perform common tasks within the simulation environment. The tasks described in this section are likely to be of particular interest to BIOS developers. However, developers of other types of software will benefit as well, especially from tasks like logging CPU cycles and using the debugger.

14.1 Loading a BIOS Image

1. Move the BIOS ROM image into your Images directory.
2. Use "*View→Show Devices*" to show the Devices Window, shown in Figure 3-2 on page 9.
3. Right-click on the system-BIOS memory device icon in the Device Window and select the "*Configure Device*" option on the Workspace Popup Menu (Figure 3-3 on page 11).
4. Choose the "*Memory Configuration*" tab.
5. Enter the appropriate base address and size for your BIOS ROM.
6. Browse for your BIOS ROM image file. The browser will only show files that have a ROM or BIN filename extension.
7. Select the read-only option, unless the BIOS code will modify its image within the device.
8. For most BIOS ROM select the system BIOS ROM, memory-address masking, and memory is non-cacheable options.
9. Click OK to close the configuration dialog and accept the changes.

14.2 Changing DRAM Size

There are two ways to configure the simulated memory size. For generic memory size configuration in powers of two you can use the *Memory Configurator*, see Figure 14-1 and for specific or non-symmetric DIMM configurations please follow the steps on page 168.

To open the *Memory Configurator* dialog click on the main menu item *View* and then choose *Show Memory Configurator* (*View→Show Memory Configurator*).

The *Memory Configurator* populates each DIMM device with two DIMMs of all identical size and type. It accounts for DDR and DDR2 and registered or unregistered memory types as required. The SPD files are loaded using the default path for SPD files "./Images/<spdfile>".

Please be advised that memory configurations that are too large will slow down the simulation significantly and may also confuse some BIOS's.

Note: The public release of the simulator supports only up to 4 GB of simulated memory.

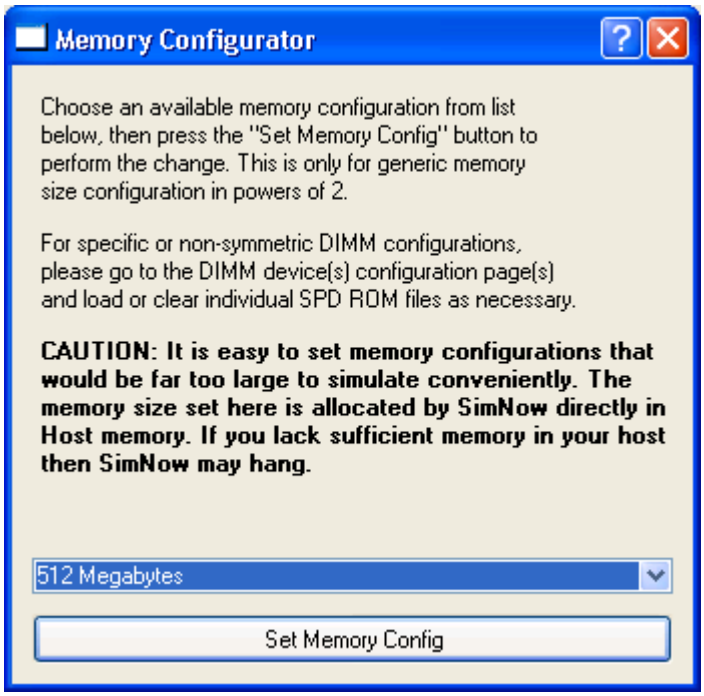

Memory Configurator
Choose an available memory configuration from list below, then press the "Set Memory Config" button to perform the change. This is only for generic memory size configuration in powers of 2.
For specific or non-symmetric DIMM configurations, please go to the DIMM device(s) configuration page(s) and load or clear individual SPD ROM files as necessary.
CAUTION: It is easy to set memory configurations that would be far too large to simulate conveniently. The memory size set here is allocated by SimNow directly in Host memory. If you lack sufficient memory in your host then SimNow may hang.
512 Megabytes
Set Memory Config

Figure 14-1: Memory Configurator

Note: The public release of the simulator supports no specific or non-symmetric DIMM configurations. To change the simulated memory size please use the Memory Configurator.

If you want specific or non-symmetric DIMM configurations please follow these steps:

1. Use "*View→Show Devices*" to show the Devices Window.
2. Right-click on the DIMM-memory device icon in the Device Window and select the "*Configure Device*" option on the Workspace Popup Menu (Figure 3-3 on page 11).
3. Select the tab for the DIMM slot that you wish to alter.
4. Click the Import SPD button and browse for an appropriate SPD file. The SPD files should be stored in the Images directory. The SPD filename should give an indication of the size of the DIMM that it represents.
5. A DIMM can be eliminated from the system, by changing the contents of SPD byte 0 (Number of SPD Bytes Used) to zero.
6. Click OK to close the configuration property sheet and accept the changes.

14.3 Changing SPD Data

Any byte of SPD data can be altered in order to model DIMM configurations that do not currently exist. The process for modifying a SPD data byte is as follows:

1. Use "*View→Show Devices*" to show the Devices Window.
2. Right-click on the DIMM Memory device icon in the Device Window and select the "*Configure Device*" option on the Workspace Popup Menu (Figure 3-3 on page 11).

3. Select the tab for the DIMM slot that you wish to alter.
4. Select an SPD byte description from the large list box. The corresponding data byte will be shown as two hex digits in the small edit box to the right of the list box.
5. Type a new hex value in the edit box.
6. Optionally, the altered SPD data can be saved to a file by clicking the Export SPD button.
7. Click OK to close the configuration property sheet and accept the changes.

If the contents of SPD byte 0 (Number of SPD Bytes Used) is set to zero, the DIMM will not respond to any SMBUS accesses. This allows simulation of a DIMM module that does not include an SPD ROM.

14.4 Clearing CMOS

View the Devices Window and double-click on the Southbridge. Choose the "*CMOS"* tab.

1. Save the current CMOS to disk and call it "*blank.cmos*".
2. Open the file in Notepad and change all the data fields from their current values to the desired fill pattern (usually 0x00 or 0xFF; do not include the *h* character in the file). Save the file. These first three steps are needed only once.
3. Reload the file into the simulator whenever you wish to clear CMOS.
4. View the Diagnostic Port Output in the Main Window, as shown in Figure 14-2.

Figure 14-2: Diagnostics Display

The Diagnostic Display displays data written to three I/O address ranges, 0x80-0x83, 0x84-0x87, 0xE0-0xE3. Currently, the Diagnostic Display is implemented only for Southbridge device. If the system configuration includes a Southbridge device, then the Diagnostic Display will be displayed.

14.5 Logging PCI Configuration Cycles

Northbridge devices can be configured to produce PCI configuration-cycle log messages. Complete the following steps to enable and capture of these log messages.

1. Open the Device Window from the Main Window Menu ("*View→Show Devices"*). Double-click on the Northbridge device. This will bring up the device Properties Window. Click on *Logging Capabilities* that will display the logging options. Select *Log PCI Configuration Cycle* to and then click OK to accept the configuration.
2. Select "*View→Log Window*" from the Main Window Menu. This will bring up a Message Log dialog box similar to the one shown in *Figure 14-3*.

3. Log messages will only be captured from devices that have a check beside their name. If the Northbridge device does not have a check, then check it by clicking its check box.
4. Select whether to send log messages to the window, and/or to a file. If logging to a file, enter a filename for the log file.
5. Execute the simulation, and the requested information will be logged.

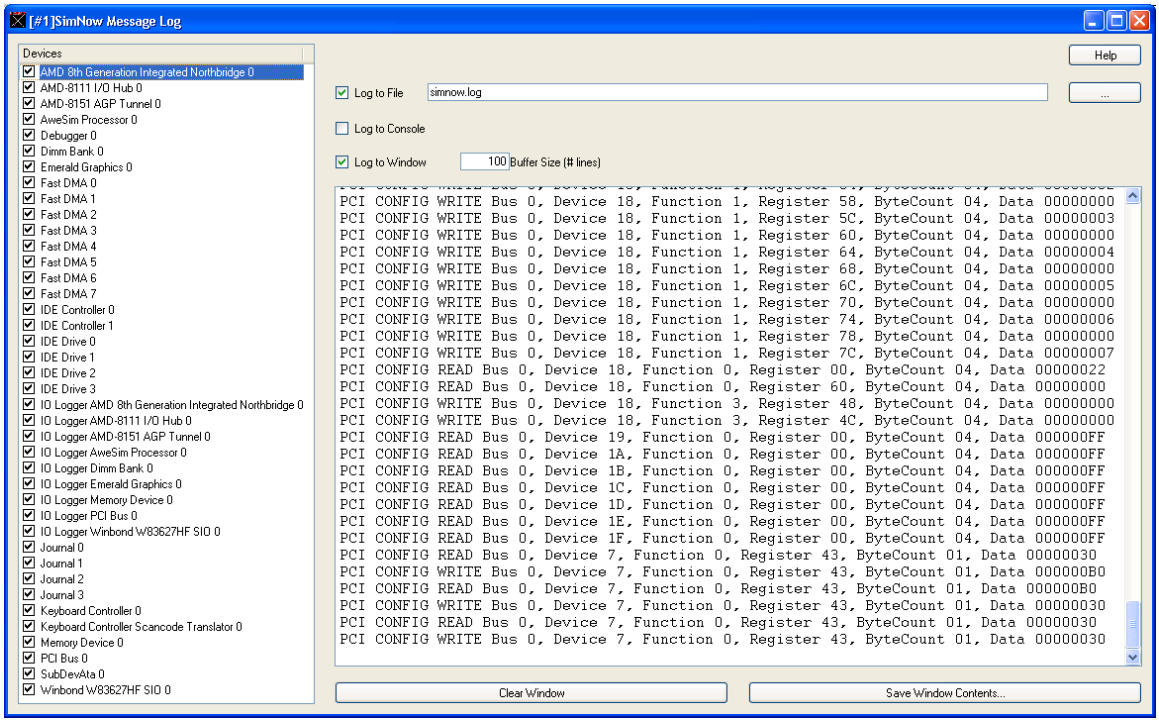

[#1]SimNow Message Log
Devices
AMD 8th Generation Integrated Northbridge 0
AMD-8111 I/O Hub 0
AMD-8151 AGP Tunnel 0
AweSim Processor 0
Debugger 0
Dimm Bank 0
Emerald Graphics 0
Fast DMA 0
Fast DMA 1
Fast DMA 2
Fast DMA 3
Fast DMA 4
Fast DMA 5
Fast DMA 6
Fast DMA 7
IDE Controller 0
IDE Controller 1
IDE Drive 0
IDE Drive 1
IDE Drive 2
IDE Drive 3
IO Logger AMD 8th Generation Integrated Northbridge 0
IO Logger AMD-8111 I/O Hub 0
IO Logger AMD-8151 AGP Tunnel 0
IO Logger AweSim Processor 0
IO Logger Dimm Bank 0
IO Logger Emerald Graphics 0
IO Logger Memory Device 0
IO Logger PCI Bus 0
IO Logger Winbond W83627HF SIO 0
Journal 0
Journal 1
Journal 2
Journal 3
Keyboard Controller 0
Keyboard Controller Scancode Translator 0
Memory Device 0
PCI Bus 0
SubDevAta 0
Winbond W83627HF SIO 0
Help
Log to File
simnow.log
Log to Console
Log to Window
100
Buffer Size (# lines)
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 58, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 5C, ByteCount 04, Data 00000003
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 60, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 64, ByteCount 04, Data 00000004
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 68, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 6C, ByteCount 04, Data 00000005
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 70, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 74, ByteCount 04, Data 00000006
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 78, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 1, Register 7C, ByteCount 04, Data 00000007
PCI CONFIG READ Bus 0, Device 18, Function 0, Register 00, ByteCount 04, Data 00000022
PCI CONFIG READ Bus 0, Device 18, Function 0, Register 60, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 3, Register 48, ByteCount 04, Data 00000000
PCI CONFIG WRITE Bus 0, Device 18, Function 3, Register 4C, ByteCount 04, Data 00000000
PCI CONFIG READ Bus 0, Device 19, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 1A, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 1B, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 1C, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 1D, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 1E, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 1F, Function 0, Register 00, ByteCount 04, Data 000000FF
PCI CONFIG READ Bus 0, Device 7, Function 0, Register 43, ByteCount 01, Data 00000030
PCI CONFIG WRITE Bus 0, Device 7, Function 0, Register 43, ByteCount 01, Data 000000B0
PCI CONFIG READ Bus 0, Device 7, Function 0, Register 43, ByteCount 01, Data 000000B0
PCI CONFIG WRITE Bus 0, Device 7, Function 0, Register 43, ByteCount 01, Data 00000030
PCI CONFIG READ Bus 0, Device 7, Function 0, Register 43, ByteCount 01, Data 00000030
PCI CONFIG WRITE Bus 0, Device 7, Function 0, Register 43, ByteCount 01, Data 00000030
Clear Window
Save Window Contents...

Figure 14-3: Message Log Window

14.6 Logging CPU Cycles

Setting up the simulator to log CPU cycles requires most of the steps detailed in Section 14.5, "*Logging PCI Configuration Cycles*". However, in this case, the messages from the CPU are captured. The steps are:

1. Open the Device Window ("*View→Show Devices*"). Double-click on the CPU device. This will bring up the device Properties Window that will list available logging options. Select the desired logging options. Click OK to accept the configuration. See Section 7.1, "*AweSim Processor Device*", on page 55 to obtain detailed information about CPU Logging options.
2. Select "*View→Log Window*" from the Main Window Menu. This will bring up a Message Log dialog box similar to the one shown in *Figure 14-3*.
3. Log messages will only be captured from devices that have a check beside their name. If the CPU device does not have a check, then check it by clicking its check box.
4. Repeat the steps here.

14.7 Creating a Floppy-Disk Image

Use the DiskTool utility to create a floppy-disk image file suitable for loading into the simulator. DiskTool is located in the "*SimNow\Tools*" directory. To create an image of a physical floppy disk, see Section 13, "DiskTool", on page 161.

When the image has been created, it can be loaded into the simulation as described in Section 5.1.1, "*Open a Simulation Definition*", on page 40.

This page is intentionally blank.

15 Frequently Asked Questions (FAQ)

Why is the mouse cursor very difficult to control inside the simulated display area?
The mouse on the Host and in the Guest do not track each other very well in general. We provide another mouse mode to help with this. Click on the menu item "Special Keyboard→Grab Mouse and Keyboard", see Section 5.2.3, "Interaction with the Simulated Machine", on page 45.

Please note that this mode has interaction issues with the Exceed X-server on Windows if you're running a Linux hosted version of the simulator and displaying it over a network to a Windows PC desktop.

Why does the on-line help not work on Linux?
Quit any local Mozilla browsers before clicking on the on-line help menu items or buttons in the simulator.

What is SimNow™ software?
See Section 1, "Overview", on page 1.

Is SimNow faster than my old Vax 780?
See Section 1, "Overview", on page 1.

What is a "BSD" file?
See Section 6.1, "BSD Files", on page 49.

What do you need to run the simulator?
See Section 2, "Installation", on page 3.

What generic BSD files are provided with the simulator?
See Section A.2.1, "Computer Platform Files", on page 178.

How do I load a BSD file?
See Section 5.1.1, "Open a Simulation Definition File", on page 40.

How do I Start, Stop, Reset, Press Soft Sleep, or Press Soft Power for simulations?
See Section 3.1, "Tool Bar Buttons", on page 7.

What kind of hardware does the simulator require?
See Section 2.1, "System Requirements", on page 3.

What host operating systems can the simulator be run on?
See Section 2.1, "System Requirements", on page 3.

What Guest operating systems are supported?
See Section A.3, "Supported Guest Operating Systems", on page 180.

What devices are supported?
See Section 7, "Device Configuration", on page 53.

What about graphics/video adapter?
See Section 1, "Overview", on page 1 and Section 7.4, "Emerald Graphics Device on page 65.

What about networking?
See Section 7.24, "E1000 Network Adapter Device", on page 124.

How does the simulator access media? What are Hard Disk, DVD-/ CD-ROM Disk, or Floppy Disk images?
See Section 4, "Disk Images", on page 35.

How do I create Disk images? What is DiskTool?
See Section 4, "Disk Images", on page 35.

How do I attach to a Hard Disk, DVD-/CD-ROM Disk, or Floppy Disk image?
All three kinds of images, including blank Hard Disk images of the desired size, can be created on both Windows 64 Beta and Linux-64 Hosts with our DiskTool program provided in the simulator release package.

The usage is relatively self-explanatory from its GUI, and it can also be run from the command-line. Check out the command-line options via "DiskTool -h".

So, this file allows you to save a running simulation to a file. At any later time, you can open this file in SimNow to restore the simulation to the same point where you left off.

How do I access the integrated Debugger?
See Section 10, "CPU Debugger", on page 147.

How do I copy files into the simulator?
See Section 5.2.1, "Assigning Disk-Image", on page 42.

How do I copy files out of the simulator?
See Section 5.2.1, "Assigning Disk-Image", on page 42.

Where can I find the POST codes/Diagnostic port values of the simulation?
See Section 3.4.1, "SimStats and Diagnostic Ports", on page 29.

How do I edit device configurations in SimNow?
See Section 3.2, "Device Window", on page 9.

How do I change a BIOS in a BSD?
See Section 7.7, "Memory Device - Configuration Options", on page 81.

How do I change the amount of system RAM installed in a BSD?
See Section 7.3, "DIMM Device", on page 59.

How do I change the processor type of a processor in a BSD?
See Section 7.1, "AweSim Processor Device - Configuration Options", on page 55.

How do I enable or disable IDE Hard Disk image journaling?
See Section 5.2.1, "Assigning Disk-Image", on page 42 or A.7.2 IDE on page 229.

Why does Windows Server 2003 crash?
See Section A.3, "Supported Guest Operating Systems", on page 181.

DiskTool displays an "Operation failed!" message box.
See Section 13.2, "GUI Mode", on page 162.

Why doesn't the simulator work on Linux kernels prior to version 2.6.10?
See Section 2.1, "System Requirements", on page 3.

Why is the graphics performance in simulation so slow?
See Section 7.4, "Emerald Graphics Device - Improve Graphics Performance", on page 68.

Why doesn't the simulated Operating System correctly recognize the DVD/CD after I changed the DVD/CD image?
When changing DVD/CD images clear the old image, allow the simulation to run for a couple of seconds, and then set the new image. This gives the Operating System a chance to see that the DVD-/CD-ROM is "not ready", and it more correctly detects that the DVD/CD image has changed. For example:

<press "Stop" button>
```
1 simnow>ide:1.image 0 off
```
<press "go" button>
<wait 5 seconds>
<press "Stop" button>
```
1 simnow>ide:1.image 0 c:\fc3-x86_64-disc2.iso
```

The serial connection to Microsoft's KLernel Debugger seems to be unstable. What can I do?
See Section 11.1, "Kernel Debugger", on page 155.

How can I obtain the full release version of the simulator?
See Section 1, "Overview", on page 1.

Why doesn't the OS find a connected USB device?
The USB port may not be soft-enabled. For example to soft-enable USB port:

```
1 simnow> usb:0.Port enable 0
```

A Appendix

A.1 Format of Floppy and Hard-Drive Images

The floppy-disk format assumes a standard 1.44 Mbyte floppy disk, consisting of 80 cylinders, 2 heads, and eighteen 512-byte sectors per head (36 sectors per cylinder). The image file consists simply of each sector, starting with the first sector of the first cylinder on the first head, and proceeding sequentially through the last sector of the last cylinder on the second head. The total size of the image file is identical to the total capacity of a 1.44 Mbyte floppy disk, or 1,474,560 bytes.

The hard-disk image is formatted in a similar fashion, with the exception that the total number of cylinders, heads, and sectors per head varies. Because of this, the hard-disk image file contains a 512-byte header before the raw data. This 512-byte header is identical to the information provided by the drive in response to the ATA command "*IDENTIFY*". Following the 512-byte header is the data for each sector from the device. As with the floppy, the data starts with the first sector of the first cylinder on the first head. Unlike floppies, however, the image file may end before the last sector of the last cylinder on the last head. If an attempt is made by the simulator, to access data on the drive image that is beyond the end of the available data (but still within the bounds defined by the geometry of the device), the simulator will extend the image file dynamically.

The BSD file contains the contents of all Viper Plus registers. It also saves the contents of any buffers and the states of all internal devices (HDD controllers, PIT, PIC, etc.). When the BSD file is read in, all buffers are filled with past data, and all states are restored to their saved states.

The symbol files that the debugger uses are in a simple text format. Each line contains an address, length, and symbol name. Any line that starts with a semicolon is considered a comment. Following is a sample file:

```
; SimNow Debugger Symbol File Format
;
; Address   Length   Symbolic Name
004011f0    3f        main
00401a3c    0         GetModuleHandleA@4
00401a42    0        _GetCommandLineA@0
```

The address value may be an absolute address or a module-relative address. For the latter case, the load address may be specified when the symbols are loaded into the debugger with the "*load_symbols*" command (see Section 10.2, "*Debugger Command Reference*", on page 151).

A.2 Bill of Material

A.2.1 Computer Platform Files (BSD)

This section gives a brief description of the computer platform description (BSD) files, devices, and disk- and ROM-image files that come with AMD SimNow™ Platform Simulator.

s

File name	# CPUs	# CPU Cores	# PCI Buses	Southbridge	SIO	Graphic Type	Public Release
Solo[1]	1	1	1	AMD-8111	W83627HF	AGP	✗
Fuge	8	1	4	AMD-8111	W83627HF	PCI	✗
Melody_1p	1	1	1	AMD-8111	W83627HF	PCI	✓
Melody_1p_jh	1	2	4	AMD-8111	W83627HF	PCI	✓
Melody_2p	2	1	4	AMD-8111	W83627HF	PCI	✓
Melody_2p_jh	2	2	4	AMD-8111	W83627HF	PCI	✓
Quartet	4	1	4	AMD-8111	W83627HF	PCI	✓
Serenade_1p-ami	1	1	3	AMD-8111	W83627HF	PCI	✗
Serenade-ami	2	2	3	AMD-8111	W83627HF	PCI	✗
Family10h_1p	1	4	3	AMD-8111	W83627HF	PCI	✓
Family10h_2p	2	4	3	AMD-8111	W83627HF	PCI	✗
Warthog2_Family10h	4	4	2	AMD-8111	W83627HF	PCI	✗
Cat2_Family11h	2	1	1	SB600	SMSC KBC1100	PCI	✗
Warthog2	4	1	1	AMD-8111	W83627HF	PCI	✓
Cheetah_1p	1	1	3	AMD-8111	W83627HF	PCI	✓
Cheetah_1p_jh	1	2	3	AMD-8111	W83627HF	PCI	✓
Cheetah_2p	2	1	3	AMD-8111	W83627HF	PCI	✓
Cheetah_2p_jh	2	2	3	AMD-8111	W83627HF	PCI	✓
Vp_bd_phase1	1	1	3	AMD-8111	W83627HF	PCI	✓
Vp_bd_phase2	1	4	3	AMD-8111	W83627HF	PCI	✗
Sahara_Family10h	1	4	1	SB400	ITE8712SIO	PCI	✗
Shiner_family10h	1	4	1	SB700	ITE8712SIO	PCI	✓
Dune	1	1	1	SB400	ITE8712SIO	PCI	✗
Drachma_peso_1p_family10h	2	6	1	SB700	W83627HF	PCI	✗

Table 15-1: Computer Platform Files (BSD)

A.2.2 Device Files (*.BSL)

Please see Section 7, "*Device Configuration*", on page 53 for device listings and descriptions.

A.2.3 Product Files (*.ID)

A product file configures the CPU and Northbridge to represent and behave as an actual AMD product. A product file will set the CPUID Family Model and Stepping, the BrandID, the MANID, and fuses.

[1] This is the recommended default uniprocessor platform.

Product File	CPU Type	# CPU Cores	PIN	Rev.	AMD Virtualization	Public Release
Athlon64-754_SH-C0_(800MHz).id	AMD Athlon64	1	754	C0	✗	✓
Athlon64-754_SH-CG_(800MHz).id	AMD Athlon64	1	754	CG	✗	✓
Athlon64-754_SH-D0_(800MHz).id	AMD Athlon64	1	754	D0	✗	✓
Athlon64-754_SH-E0_(800MHz).id	AMD Athlon64	1	754	E0	✗	✓
Athlon64-939_JH-E0_(800MHz x2).id	AMD Athlon64	2	939	E0	✗	✓
Athlon64-939_SH-CG_(800MHz).id	AMD Athlon64	1	939	CG	✗	✓
Athlon64-939_SH-D0_(800MHz).id	AMD Athlon64	1	939	D0	✗	✓
Athlon64-939_SH-E0_(800MHz).id	AMD Athlon64	1	939	E0	✗	✓
Athlon64-AM2_BH-G1B_(800MHz x2).id	AMD Athlon64	2	AM2	G1B	✓	✓
Athlon64-AM2_JH-F2G_(800MHz x2).id	AMD Athlon64	2	AM2	F2G	✓	✓
Athlon64-AM2_SH-F0_(800MHz).id	AMD Athlon64	1	AM2	F0	✓	✓
Athlon64-S1_BH-G1B_(800MHz x2).id	AMD Athlon64	2	S1	G1B	✓	✓
Athlon64-S1_JH-F2G_(800MHz x2).id	AMD Athlon64	2	S1	F2G	✓	✓
Athlon64-S1_SH-F0_(800MHz).id	AMD Athlon64	1	S1	F0	✓	✓
Opteron-940_JH-E0_(800MHz x2).id	AMD Opteron	2	940	E0	✗	✓
Opteron-940_SH-B3_(800MHz).id	AMD Opteron	1	940	B3	✗	✓
Opteron-940_SH-C0_(800MHz).id	AMD Opteron	1	940	C0	✗	✓
Opteron-940_SH-CG_(800MHz).id	AMD Opteron	1	940	CG	✗	✓
Opteron-940_SH-D0_(800MHz).id	AMD Opteron	1	940	D0	✗	✓
Opteron-940_SH-E0_(800MHz).id	AMD Opteron	1	940	E0	✗	✓
Opteron-L1_JH-F0_(800Mhz x2).id	AMD Opteron	2	L1	F0	✓	✓
Opteron-L1_JH-F2G_(800Mhz x2).id	AMD Opteron	2	L1	F2G	✓	✓
Opteron-L1_SH-F0_(800Mhz).id	AMD Opteron	1	L1	F0	✓	✓
Family10hDR-L1_A0.id	Family 10h	4	L1	A0	✓	✓
Family10hDR-L1_B0.id	Family 10h	4	L1	B0	✓	✓
Family10hDR-L1_C0.id	Family 10h	4	L1	C0	✓	✓
Family10hDR-AM2_B0.id	Family 10h	4	AM2	B0	✓	✓
Family10hRB-AM3_C0.id	Family 10h	4	AM3	C0	✓	✓
Family10hBL-AM3_C2A.id	Family 10h	4	AM3	C2A	✓	✗
Family10hHY-G3M_D0A.id	Family 10h	12 or 8	G34	D0A	✓	✗
Family10hHY-G3S_D0A.id	Family 10h	6 or 4	G34	D0A	✓	✗
Family10hHY-L1_D0A.id	Family 10h	6	L1	D0A	✓	✗
Family11h-S1_A0.id	Family 11h	2	S1	A0	✓	✗
Family11h-S1_B0.id	Family 11h	2	S1	B0	✓	✗

Table 15-2: Product Files

A.2.4 Image Files (*.HDD, *.FDD, *.ROM, *.SPD, *.BIN)

An image file is an exact representation of a media including the contents and the logical format.

A.2.4.1 Hard-Disk Image Files

Table 15-3 shows hard-disk image files present in the simulator. These images can be found in the simulators "*/image*" folder (see Section 2.3, "*Directory Structure and Executable*", on page 4).

File name	Description
Bare-4gig.hdd	4-GB bare hard disk image.
Bare-8gig.hdd	8-GB bare hard disk image.

Table 15-3: Hard-Disk Images

A.2.4.2 Memory SPD Files

When a computer is booted (started), serial presence detect (SPD) is information stored in an electrically erasable programmable read-only memory (EEPROM) chip on memory module that tells the BIOS the memory module's size, data width, and speed. The BIOS uses this information to configure the memory properly for maximum reliability and performance.

File name	Description	Public Release
simnow_DDR_32M.spd	32MB DDR memory	✗
simnow_DDR_64M.spd	64MB DDR memory	✗
simnow_DDR_128M.spd	128MB DDR memory	✗
simnow_DDR_256M.spd	256MB DDR memory	✗
simnow_DDR_512M.spd	512MB DDR memory	✗
simnow_DDR_1G.spd	1024MB DDR memory	✗
simnow_DDR_2G.spd	2048MB DDR memory	✗
simnow_DDR_4G.spd	4096MB DDR memory	✗
simnow_DDR_32M_Reg.spd	32MB registered DDR memory	✗
simnow_DDR_64M_Reg.spd	64MB registered DDR memory	✗
simnow_DDR_128M_Reg.spd	128MB registered DDR memory	✗
simnow_DDR_256M_Reg.spd	256MB registered DDR memory	✗
simnow_DDR_512M_Reg.spd	512MB registered DDR memory	✗
simnow_DDR_1G_Reg.spd	1024MB registered DDR memory	✗
simnow_DDR_2G_Reg.spd	2048MB registered DDR memory	✗
simnow_DDR_4G_Reg.spd	4096MB registered DDR memory	✗
simnow_DDR2_128M.spd	128MB DDR2 memory	✓
simnow_DDR2_256M.spd	256MB DDR2 memory	✓
simnow_DDR2_512M.spd	512MB DDR2 memory	✓
simnow_DDR2_1G.spd	1024MB DDR2 memory	✓
simnow_DDR2_2G.spd	2048MB DDR2 memory	✓
simnow_DDR2_4G.spd	4096MB DDR2 memory	✗
simnow_DDR2_8G.spd	8192MB DDR2 memory	✗
simnow_DDR2_16G.spd	16384MB DDR2 memory	✗
simnow_DDR2_128M_Reg.spd	128MB registered DDR2 memory	✓
simnow_DDR2_256M_Reg.spd	256MB registered DDR2 memory	✓
simnow_DDR2_512M_Reg.spd	512MB registered DDR2 memory	✓
simnow_DDR2_1G_Reg.spd	1024MB registered DDR2 memory	✓
simnow_DDR2_2G_Reg.spd	2048MB registered DDR2 memory	✓
simnow_DDR2_4G_Reg.spd	4096MB registered DDR2 memory	✗
simnow_DDR2_8G_Reg.spd	8192MB registered DDR2 memory	✗
simnow_DDR2_16G_Reg.spd	16384MB registered DDR2 memory	✗
IBM_512_Reg.spd	512MB registered DDR memory	✗
Smart_DDR_128_2_133.spd	128MB DDR memory	✗

Table 15-4: Memory SPD Files

In order to use *unbuffered* DDR/DDR2 memory we recommend using the "*simnow_DDRx_yyyy_.spd*" SPD files. To use *buffered* DDR/DDR2 memory use the "*simnow_DDRx_yyyy_reg.spd*" SPD files (for DDR2 x = 2 and yyyy = size in Mbytes).

A.3 Supported Guest Operating Systems

Table 15-5 lists the guest OS compatibility matrix.

Operating System	Known Issues
Windows 2000 UP	No known issues.
Windows XP (32-Bit) UP	No known issues.
Windows XP (32 Bit) MP	No known issues.
Windows XP (64-Bit) UP	No known issues.
Windows Server 2003 (32-Bit) UP	No known issues.
Windows Server 2003 (64-Bit) UP	No known issues.
Windows Server 2003 (64-Bit) MP	No known issues.
Windows Vista (32-Bit/64-Bit) UP/MP	No known issues.
Windows Server 2008	No known issues.
MS-DOS	No known issues.
Linux (32-bit/64-bit), RedHat/SuSE, UP/MP	Kernel versions 2.4 and 2.6 are all known to work.
SUSE LiveCD 9.1	Hangs during PCMCIA probe when the VESA BIOS Extension is enabled and the active VESA Mode is **not** 1024x768.
SUSE LiveCD 9.2	No known issues.
SUSE LiveCD 9.3	No support for initial graphical setup screen. Setup screen will appear in text mode.
SUSE 10.1	No known issues.
Red Hat Enterprise Linux 4	No known issues.
Solaris x86	No known issues.
Solaris 10 for AMD64	No known issues.

Table 15-5: Supported Guest Operating Systems

The simulator has recently (but not specifically tested for this release):

- Successfully completed a 64-bit SpecJBB run on a simulated 4-processor machine. The simulator has also successfully completed the entire SPECint2000 and SPECfp2000 suite.
- Successfully completed an in-memory run of TPC-C on a simulated multi-processor system, as well as parts of TPC-C on a simulated RAID device.
- Successfully completed Sysmark® 2004's Office Productivity section and parts of Internet Content Creation.

A.4 CPUID

This section is an overview of the CPUID feature implementation in the AweSim CPU processor model.

A.4.1 CPUID Standard Feature Support (Standard Function 0x01)

Table 15-6 shows the standard feature bits returned by the AweSim CPU processor model and which features are fully (✔) or only partially (⚠) implemented and supported. A ✖ indicates that the returned feature bit is zero and this feature is not implemented and not supported.

Feature	7th Generation	8th Generation (Base)	8th Generation Pre.-Rev. F	8th Generation Rev. F
Floating-Point Unit	✔	✔	✔	✔
Virtual Mode Extensions	✔	✔	✔	✔
Debugging Extensions[1]	⚠	⚠	⚠	⚠
Page-Size Extension	✔	✔	✔	✔
Time Stamp Counter	✔	✔	✔	✔
AMD Model-Specific Registers	✔	✔	✔	✔
Physical-Address Extensions	✔	✔	✔	✔
Machine Check Exception	✔	✔	✔	✔
CMPXCHG8B Instruction	✔	✔	✔	✔
APIC	✔	✔	✔	✔
SYSENTER and SYSEXIT	✔	✔	✔	✔
Memory Type Range Registers	✔	✔	✔	✔
Page Global Extension	✔	✔	✔	✔
Machine Check Architecture	✔	✔	✔	✔
Conditional Move Instruction	✔	✔	✔	✔
Page Attribute Table	✔	✔	✔	✔
Page Size Extensions (PSE-36)	✔	✔	✔	✔
CFLUSH Instruction	✖	✔	✔	✔
MMX™ Instructions	✔	✔	✔	✔
FXSAVE/FXRSTOR	✔	✔	✔	✔
SSE	✔	✔	✔	✔
SSE2	✔	✔	✔	✔
Hyper Threading	✖	✖	✖	✖
SSE3/PNI	✖	✖	✖	✔
Monitor/MWAIT	✖	✖	✖	✖

Table 15-6: CPUID Standard Feature implementation

A.4.2 CPUID AMD Feature Support (Extended Function 0x80000001)

Table 15-7 shows the extended feature bits returned by the AweSim CPU processor model and which features are fully (✔) or only partially (⚠) implemented and

[1] Only read and write to debug registers is supported, side affects are not implemented.

supported. A ✗ indicates that the returned feature bit is zero and this feature is not implemented and not supported.

Feature	7th Generation	8th Generation (Base)	8th Generation Pre.-Rev. F	8th Generation Rev. F
Floating-Point Unit	✓	✓	✓	✓
Virtual Mode Extensions	✓	✓	✓	✓
Debugging Extensions[1]	⚠	⚠	⚠	⚠
Page-Size Extension	✓	✓	✓	✓
Time Stamp Counter	✓	✓	✓	✓
AMD Model-Specific Registers	✓	✓	✓	✓
Page Address Extensions	✓	✓	✓	✓
Machine Check Exception	✓	✓	✓	✓
CMPXCHG8B Instruction	✓	✓	✓	✓
APIC	✓	✓	✓	✓
SYSCALL and SYSRET	✓	✓	✓	✓
Memory Type Range Registers	✓	✓	✓	✓
Page Global Extension	✓	✓	✓	✓
Machine Check Architecture	✓	✓	✓	✓
Conditional Move Instruction	✓	✓	✓	✓
Page Attribute Table	✓	✓	✓	✓
Page Size Extensions (PSE-36)	✓	✓	✓	✓
No-execute page protection	✓	✓	✓	✓
SEM[2]	✗	✗	✗	✗
AMD extensions to MMX™	✓	✓	✓	✓
MMX™	✓	✓	✓	✓
FXSAVE/FXRSTOR	✓	✓	✓	✓
Fast FXSAVE/FXRSTOR	✗	✗	✗	✗
1 GB Paging feature	✗	✗	✗	✗
RDTSCP	✗	✗	✗	✗
Long Mode[2]	✗	✓	✓	✓
AMD Extensions to 3DNow!™	✓	✓	✓	✓
3DNow! Instructions	✓	✓	✓	✓
Virtualization Technology	✗	✗	✗	✓

Table 15-7: CPUID Extended Feature implementation

[1] Only read and write to debug registers is supported, side effects are not implemented.
[2] Controlled by FUSE state.

A.5 Known Issues

A.5.1 FSAVE/FRSTOR and FSTENV/FLDENV

When the simulator is executing FSAVE/FRSTOR and FSTENV/FLDENV in real-mode it is using the 16-bit protected-mode memory format.

A.5.2 Triple Faulting

If the processor encounters an exception while trying to invoke the double fault (#DF) exception handler, a triple fault exception occurs. This can rarely occur, but is possible. For example, if the invocation of a double fault exception causes the stack to overflow, then this would cause a triple fault. When this happens, the CPU will triple fault and cause a shutdown-cycle to occur. This special cycle should be interpreted by the motherboard hardware, which then pulls RESET, which ultimately resets the CPU and the computer.

However, the simulator does not simulate triple faults. In case a triple fault occurs, the simulator stops the simulation. The simulation cannot be restarted until a reset is asserted but the simulation state can be inspected with the simulator's debugger.

A.5.3 Performance-Monitoring Counter Extensions

Setting CR4.PCE (bit 8) to 1 allows software running at any privilege level to use the RDPMC instruction. Software uses the RDPMC instruction to read the four performance-monitoring MSRs, PerfCTR[3:0]. Clearing PCE to 0 allows only the most-privileged software (CPL=0) to use the RDPMC instruction.

The simulator does support the RDPMC instruction but there is no logic behind the simulated performance counter MSRs.

A.5.4 Microcode Patching

Microcode patches do not affect the simulated machine behavior. This may have unintended consequences.

A.5.5 Instruction Coherency

Instruction coherency does not work when code pages have multiple virtual-mappings. Here is an example that would not work right:

1. There is an x86 physical page that has code on it.
2. This page is mapped by two different virtual addresses, A and B
3. There is a store to virtual page A
4. We execute code from page B
5. We store again to A, modifying some of the x86 code that we executed in step 4
6. We execute the code from step 4 again

The code we execute in step 6 will probably be the old code because our page-based coherency mechanism depends on the software TLB to write protect pages which have x86 code that has been translated. However, this mechanism protects physical pages through the virtual mapping mechanism and this mechanism only knows about one virtual address mapping, not all possible mappings of any code page.

A.6 Instruction Reference

This section specifies the hexadecimal and/or binary encodings for the opcodes that SimNow does (✔), does not (✘) or does partially (⚠) simulate when simulating an AMD 8th Generation CPU, Rev. F.

A.6.1 Notation

A.6.1.1 Mnemonic Syntax

Each instruction has a syntax that includes the mnemonic and any operands that the instruction can take. Figure A-1 shows an example of a syntax in which the instruction takes two operands. In most instruction that take two operands, the first (left-most) operand is both a source operand (the first source operand) and the destination operand. The second (right-most) operand serves only as a source, not a destination.

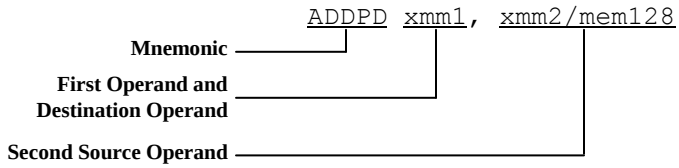


Figure A-1: Syntax for Typical Two-Operand Instruction

The following notation is used to denote the size and type of source and destination operands:

- *cReg* – Control Register.
- *dReg* – Debug register.
- *imm8* – Byte (8-Bit) immediate.
- *imm16* – Word (16-Bit) immediate.
- *imm16/32* – Word (16-bit) or doubleword (32-bit) immediate.
- *imm32* – Doubleword (32-bit) immediate.
- *imm32/64* – Doubleword (32-bit) or quadword (64-bit) immediate.
- *imm64 – Quadword (64-bit) immediate.*
- *mem* – An operand of unspecified size in memory.
- *mem8* – Byte (8-bit) operand in memory.
- *mem16* – Word (16-bit) operand in memory.
- *mem16/32* – Word (16-bit) or doubleword (32-bit) operand in memory.
- *mem32* – Doubleword (32-bit) operand in memory.
- *mem32/48* – Doubleword (32-bit) or 48-bit operand in memory.
- *mem48* – 48-bit operand in memory.
- *mem64* – Quadword (64-bit) operand in memory.
- *mem16:16* – Two sequential word (16-bit) operands in memory.
- *mem16:32* – A doubleword (32-bit) operand followed by a word (16-bit) operand in memory.
- *mem32real* – Single precision (32-bit) floating-point operand in memory.
- *mem32int* – Doubleword (32-bit) integer operand in memory.

- *mem64real* – Double-precision (64-bit) floating-point operand in memory.
- *mem64int* – Quadword (64-bit) integer operand in memory.
- *mem80real* – Double-extended-precision (80-bit) floating-point operand in memory.
- *mem80dec* – 80-bit packed BCD operand in memory, containing 18 4-bit BCD digits.
- *mem2env* – 16-bit x87 control word or x87 status word.
- *mem14/28env* – 14-byte or 28-byte x87 environment. The x87 environment consists of the x87 control word, x87 status word, x87 tag word, last non-control instruction pointer, last data pointer, and opcode of the last non-control instruction completed.
- *mem94/108env* – 94-byte or 108-byte x87 environment and register stack.
- *mem512env* – 512-byte environment for 128-bit media, 64-bit media, and x87 instructions.
- *mmx* – Quadword (64-bit) operand in an MMX™ register.
- *mmx1* – Quadword (64-bit) operand in an MMX register, specified as the left-most (first) operand in the instruction syntax.
- *mmx2* - Quadword (64-bit) operand in an MMX register, specified as the right-most (second) operand in the instruction syntax.
- *mmx/mem32* – Doubleword (32-bit) operand in an MMX register or memory.
- *mmx/mem64* – Quadword (64-bit) operand in an MMX register or memory.
- *mmx1/mem64* - Quadword (64-bit) operand in an MMX register or memory, specified as the left-most (first) operand in the instruction syntax.
- *mmx2/mem64* - Quadword (64-bit) operand in an MMX register or memory, specified as the right-most (second) operand in the instruction syntax.
- *moffset* – Memory offset of unspecified size.
- *moffset8* – Operand in memory located at the specified byte (8-bit) offset from the instruction pointer.
- *moffset16* - Operand in memory located at the specified word (16-bit) offset from the instruction pointer.
- *moffset32* - Operand in memory located at the specified doubleword (32-bit) offset from the instruction pointer.
- *pntr16:16* – Far pointer with 16-bit selector and 16-bit offset.
- *pntr16:32* - Far pointer with 16-bit selector and 32-bit offset.
- *reg* – Operand of unspecified size in a GPR register.
- *reg8* – Byte (8-bit) operand in a GPR register.
- *reg16* – Word (16-bit) operand in a GPR register.
- *reg16/32* - Word (16-bit) or doubleword (32-bit) operand in a GPR register.
- *reg32* – Doubleword (32-bit) operand in a GPR register.
- *reg64* - Quadword (64-bit) operand in a GPR register.
- *reg/mem8* – Byte (8-bit) operand in a GPR register or memory.
- *reg/mem16* – Word (16-bit) operand in a GPR register or memory.
- *reg/mem32* – Doubleword (32-bit) operand in a GPR register or memory.
- *reg/mem64* – Quadword (64-bit) operand in a GPR register or memory.

- *rel8off* – Relative address in the current code segment, in 8-bit offset range.
- *rel16off* - Relative address in the current code segment, in 16-bit offset range.
- *rel32off* - Relative address in the current code segment, in 32-bit offset range.
- *segReg or sReg* – Word (16-bit) operand in a segment register.
- *ST(0)* – x87 stack register 0.
- *ST(i)* – x87 stack register *i*, where *i* is between 0 and 7.
- *xmm* – Double quadword (128-bit) operand in an XMM register.
- *xmm1* – Double quadword (128-bit) operand in an XMM register, specified as the left-most (first) operand in the instruction syntax..
- *xmm2* – Double quadword (128-bit) operand in an XMM register, specified as the right-most (second) operand in the instruction syntax.
- *xmm/mem64* – Quadword (64-bit) operand in a 128-bit XMM register or memory.
- *xmm/mem128* – Double quadword (128-bit) operand in a 128-bit operand in an XMM register or memory.
- *xmm1/mem128* – Double quadword (128-bit) operand in a 128-bit operand in an XMM register or memory, specified as the left-most (first) operand in the instruction syntax..
- *xmm2/mem128* – Double quadword (128-bit) operand in a 128-bit operand in an XMM register or memory, specified as the right-most (second) operand in the instruction syntax.

A.6.1.2 Opcode Syntax

In addition to the notation shown in above in "Mnemonic Syntax" on page 186, the following notation indicates the size and type of operands in the syntax of instruction syntax.

- */digit* – Indicates that the ModRM byte specifies only one register or memory (r/m) operand. The digit is specified by the ModRM reg field and is used as an instruction-opcode extension. Valid digit values range from 0 to 7.
- */r* – Indicates that the ModRM byte specifies both a register and operand and a *reg/mem* (register or memory) operand.
- *cb, cw, cd ,cp* – Specified a code-offset value and possibly a new code-segment register value. The value following the opcode is either one byte (cb), two bytes (cw), four bytes (cd), or six bytes (cp).
- *ib, iw, id* – Specifies an immediate-operand value. The opcode determines whether the value is signed or unsigned. The value following the opcode, ModRM, or SIB byte is either one byte (ib), two bytes (iw), or four bytes (id). Word and doubleword values start wit the low-order byte.
- *+rb, +rw, +rd, +rq* – Specifies a register value that is added to the hexadecimal byte on the left, forming a one-byte opcode. The result is an instruction that operates on the register specified by the register code. Valid register-code values are shown in "*AMD x86-64 Architecture: Programmer's Manual, Volume 3*".
- *m64* – Specifies a quadword (64-bit) operand in memory.

- *+i* – Specifies an x87 floating-point stack operand, ST(*i*). The value is used only with x87 floating-point instructions. It is added to the hexadecimal byte on the left, forming a one-byte opcode. Valid values range from 0 to 7.

A.6.2 General Purpose Instructions

This chapter describes the function, mnemonic syntax, and opcodes that the simulator simulates. General-purpose instructions are used in basic software execution. Most of these instructions load, store, or operate on data location in the general-purpose registers (GPRs), in memory, or in both. The remaining instructions are used to alter the sequential flow of the program by branching to other locations within the program, or to entirely different programs.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
AAA	37	Create an unpacked BCD number.	✔
AAD	D5	Adjust two BCD digits in AL and AH.	✔
AAM	D4	Create a pair of unpacked BCD values in AH and AL.	✔
AAS	3F	Create an unpacked BCD number from the contents of the AL register.	✔
ADC AL,*imm8*	14 *ib*	Add *imm8* to AL + CF.	✔
ADC AL,*imm16*	14 *iw*	Add *imm16* to AX + CF.	✔
ADC EAX,*imm32*	15 *id*	Add *imm32* to EAX + CF.	✔
ADC RAX,*imm32*	15 *id*	Add sign-ext. *imm32* to RAX + CF.	✔
ADC *reg/mem8,imm8*	80 /2 *ib*	Add *imm8* to *reg/mem8* + CF.	✔
ADC *reg/mem16,imm16*	81 /2 *iw*	Add *imm16* to *reg/mem16* + CF.	✔
ADC *reg/mem32,imm32*	81 /2 *id*	Add *imm32* to *reg/mem32* + CF.	✔
ADC *reg/mem64,imm32*	81 /2 *id*	Add sign-ext. *imm32* to *reg/mem64* + CF.	✔
ADC *reg/mem16,imm8*	83 /2 *ib*	Add sign-ext. *imm8* to *reg/mem16* + CF.	✔
ADC *reg/mem32,imm8*	83 /2 *ib*	Add sign-ext. *imm8* to *reg/mem32* + CF.	✔
ADC *reg/mem64,imm8*	83 /2 *ib*	Add sign-ext. *imm8* to *reg/mem64* + CF.	✔
ADC *reg/mem8,reg8*	10 /r	Add *reg8* to *reg/mem8* + CF.	✔
ADC *reg/mem16,reg16*	11 /r	Add *reg16* to *reg/mem16* + CF.	✔
ADC *reg/mem32,reg32*	11 /r	Add *reg32* to *reg/mem32* + CF.	✔
ADC *reg/mem64,reg64*	11 /r	Add *reg64* to *reg/mem64* + CF.	✔
ADC *reg8,reg/mem8*	12 /r	Add *reg/mem8* to *reg8* + CF.	✔
ADC *reg16,reg/mem16*	13 /r	Add *reg/mem16* to *reg16* + CF.	✔
ADC *reg32,reg/mem32*	13 /r	Add *reg/mem32* to *reg32* + CF.	✔
ADC *reg64,reg/mem64*	13 /r	Add *reg/mem64* to *reg64* + CF.	✔
ADD AL,*imm8*	04 *ib*	Add *imm8* to AL.	✔
ADD AX,*imm16*	05 *iw*	Add *imm16* to AX.	✔
ADD EAX,*imm32*	05 *id*	ADD *imm32* to EAX.	✔
ADD RAX,*imm64*	05 *id*	ADD *imm64* to RAX.	✔
ADD *reg/mem8,imm8*	80 /0 *ib*	Add *imm8* to *reg/mem8*.	✔
ADD *reg/mem16,imm16*	81 /0 *iw*	Add *imm16* to *reg/mem16*.	✔
ADD *reg/mem32,imm32*	81 /0 *id*	Add *imm32* to *reg/mem32*.	✔
ADD *reg/mem64,imm32*	81 /0 *id*	Add sign-ext. *imm32* to *reg/mem64*.	✔
ADD *reg/mem16,imm8*	83 /0 *ib*	Add sign-ext. *imm8* to *reg/mem16*.	✔
ADD *reg/mem32,imm8*	83 /0 *ib*	Add sign-ext. *imm8* to *reg/mem32*.	✔
ADD *reg/mem64,imm8*	83 /0 *ib*	Add sign-ext. *imm8* to *reg/mem64*.	✔
ADD *reg/mem8,reg8*	00 /r	Add *reg8* to *reg/mem8*.	✔
ADD *reg/mem16,reg16*	01 /r	Add *reg16* to *reg/mem16*.	✔
ADD *reg/mem32,reg32*	01 /r	Add *reg32* to *reg/mem32*.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
ADD *reg/mem64,reg64*	01 /r	Add *reg64* to *reg/mem64*.	✔
ADD *reg8,reg/mem8*	02 /r	Add *reg/mem8* to *reg8*.	✔
ADD *reg16,reg/mem16*	03 /r	Add *reg/mem16* to *reg16*.	✔
ADD *reg32,reg/mem32*	03 /r	Add *reg/mem32* to *reg32*.	✔
ADD *reg64,reg/mem64*	03 /r	Add *reg/mem64* to *reg64*.	✔
AND AL,*imm8*	24 *ib*	AND the contents of AL with an immediate 8-bit value and store the result in AL.	✔
AND AX,*imm16*	25 *iw*	AND the contents of AX with an immediate 16-bit value and store the result in AX.	✔
AND EAX,*imm32*	25 *id*	AND the contents of EAX with an immediate 32-bit value and store the result in EAX.	✔
AND RAX,*imm32*	25 *id*	AND the contents of RAX with a sign-extended immediate 32-bit value and store the result in RAX.	✔
AND *reg/mem8,imm8*	80 /4 *ib*	AND the contents of *reg/mem8* with *imm8*.	✔
AND *reg/mem16,imm16*	81 /4 *iw*	AND the contents of *reg/mem16* with *imm16*.	✔
AND *reg/mem32,imm32*	81 /4 *id*	AND the contents of *reg/mem32* with *imm32*.	✔
AND *reg/mem64,imm32*	81 /4 *id*	AND the contents of *reg/mem64* with a sign-extended *imm32*.	✔
AND *reg/mem16,imm8*	83 /4 *ib*	AND the contents of *reg/mem16* with a sign-extended 8-bit value.	✔
AND *reg/mem32,imm8*	83 /4 *ib*	AND the contents of *reg/mem32* with a sign-extended 8-bit value.	✔
AND *reg/mem64,imm8*	83 /4 *ib*	AND the contents of *reg/mem64* with a sign-extended 8-bit value.	✔
AND *reg/mem8,reg8*	20 /r	AND the contents of an 8-bit register or memory location with the contents of an 8-bit register.	✔
AND *reg/mem16,reg16*	21 /r	AND the contents of a 16-bit register or memory location with the contents of a 16-bit register.	✔
AND *reg/mem32,reg32*	21 /r	AND the contents of a 32-bit register or memory location with the contents of a 32-bit register.	✔
AND *reg/mem64,reg64*	21 /r	AND the contents of a 16-bit register or memory location with the contents of a 16-bit register.	✔
AND *reg8,reg/mem8*	22 /r	AND the contents of an 8-bit register with the contents of an 8-bit memory location or register.	✔
AND *reg16,reg/mem16*	23 /r	AND the contents of a 16-bit register with the contents of a 16-bit memory location or register.	✔
AND *reg32,reg/mem32*	23 /r	AND the contents of a 32-bit register with the contents of a 32-bit memory location or register.	✔
AND *reg64,reg/mem64*	23 /r	AND the contents of a 64-bit register with the contents of a 64-bit memory location or register.	✔
BOUND *reg16*,mem*16&mem16*	62 /r	Test whether a 16-bit array index is within the bounds specified by the two 16-bit values in *mem16&mem16*.	✔
BOUND *reg32*,mem32*&mem32*	62 /r	Test whether a 32-bit array index is within the bounds specified by the two 32-bit values in *mem32&mem32*.	✔
BSF *reg16,reg/mmem8*	0F BC /r	Bit scan forward on the contents of *reg/mem16*.	✔
BSF *reg32,reg/mmem32*	0F BC /r	Bit scan forward on the contents of *reg/mem32*.	✔
BSF *reg64,reg/mmem64*	0F BC /r	Bit scan forward on the contents of *reg/mem64*.	✔
BSR *reg16,reg/mmem8*	0F BD /r	Bit scan reverse on the contents of *reg/mem16*.	✔
BSR *reg32,reg/mmem32*	0F BD /r	Bit scan reverse on the contents of *reg/mem32*.	✔
BSR *reg64,reg/mmem64*	0F BD /r	Bit scan reverse on the contents of *reg/mem64*.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
BSWAP *reg32*	0F C8 *+rd*	Reverse the byte order of *reg32*.	✔
BSWAP *reg64*	0F C8 *+rd*	Reverse the byte order of *reg64*.	✔
BT *reg/mem16,reg16*	0F A3 /r	Copy the value of the selected bit to the carry flag.	✔
BT *reg/mem32,reg32*	0F A3 /r	Copy the value of the selected bit to the carry flag.	✔
BT *reg/mem64,reg64*	0F A3 /r	Copy the value of the selected bit to the carry flag.	✔
BT *reg/mem16,imm8*	0F BA /4 *ib*	Copy the value of the selected bit to the carry flag.	✔
BT *reg/mem32,imm8*	0F BA /4 *ib*	Copy the value of the selected bit to the carry flag.	✔
BT *reg/mem64,imm8*	0F BA /4 *ib*	Copy the value of the selected bit to the carry flag.	✔
BTC *mem/mem16,reg16*	0F BB /r	Copy the value of the selected bit to the carry flag, and then complement the selected bit.	✔
BTC *mem/mem32,reg32*	0F BB /r	Copy the value of the selected bit to the carry flag, and then complement the selected bit.	✔
BTC *mem/mem64,reg64*	0F BB /r	Copy the value of the selected bit to the carry flag, and then complement the selected bit.	✔
BTC *reg/mem16,imm8*	0F BA /7 *ib*	Copy the value of the selected bit to the carry flag, and then complement the selected bit.	✔
BTC *reg/mem32,imm8*	0F BA /7 *ib*	Copy the value of the selected bit to the carry flag, and then complement the selected bit.	✔
BTC *reg/mem64,imm8*	0F BA /7 *ib*	Copy the value of the selected bit to the carry flag, and then complement the selected bit.	✔
BTR *reg/mem16,reg16*	0F B3 /r	Copy the value of the selected bit to the carry flag, and then clear the selected bit.	✔
BTR *reg/mem32,reg32*	0F B3 /r	Copy the value of the selected bit to the carry flag, and then clear the selected bit.	✔
BTR *reg/mem64,reg64*	0F B3 /r	Copy the value of the selected bit to the carry flag, and then clear the selected bit.	✔
BTR *reg/mem16,imm8*	0F BA /6 *ib*	Copy the value of the selected bit to the carry flag, and then clear the selected bit.	✔
BTR *reg/mem32,imm8*	0F BA /6 *ib*	Copy the value of the selected bit to the carry flag, and then clear the selected bit.	✔
BTR *reg/mem64,imm64*	0F BA /6 *ib*	Copy the value of the selected bit to the carry flag, and then clear the selected bit.	✔
BTS *reg/mem16,reg16*	0F AB /r	Copy the value of the selected bit to the carry flag, and then set the selected bit.	✔
BTS *reg/mem32,reg32*	0F AB /r	Copy the value of the selected bit to the carry flag, and then set the selected bit.	✔
BTS *reg/mem64,reg64*	0F AB /r	Copy the value of the selected bit to the carry flag, and then set the selected bit.	✔
BTS *reg/mem16*,imm8	0F BA /5 *ib*	Copy the value of the selected bit to the carry flag, and then set the selected bit.	✔
BTS *reg/mem32*,imm8	0F BA /5 *ib*	Copy the value of the selected bit to the carry flag, and then set the selected bit.	✔
BTS *reg/mem64*,imm8	0F BA /5 *ib*	Copy the value of the selected bit to the carry flag, and then set the selected bit.	✔
CALL *rel16off*	E8 *iw*	Near call with the target specified by a 16-bit relative displacement.	✔
CALL *rel32off*	E8 *id*	Near call with the target specified by a 32-bit relative displacement.	✔
CALL *reg/mem16*	FF /2	Near call with the target specified by *reg/mem16*.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
CALL *reg/mem32*	FF /2	Near call with the target specified by *reg/mem32*.	✔
CALL *reg/mem64*	FF /2	Near call with the target specified by *reg/mem64*.	✔
CALL FAR *pntr16:16*	9A *cd*	Far call direct, with the target specified by a far pointer contained in the instruction.	✔
CALL FAR *pntr16:32*	9A *cp*	Far call direct, with the target specified by a far pointer contained in the instruction.	✔
CALL FAR *mem16:16*	FF /3	Far call indirect, with the target specified by a far pointer in memory.	✔
CALL FAR *mem16:32*	FF /3	Far call indirect, with the target specified by a far pointer in memory.	✔
CBW	98	Sign-extend AL into AX.	✔
CWDE	98	Sign-extend AX into EAX.	✔
CDQE	98	Sign-extend EAX into RAX.	✔
CWD	99	Sign-extend AX into DX:AX.	✔
CDQ	99	Sign-extend EAX into EDX:EAX.	✔
CQO	99	Sign-extend RAX into RDX:RAX.	✔
CLC	F8	Clear the carry flag (CF) to zero.	✔
CLD	FC	Clear the direction flag (DF) to zero.	✔
CFLUSH *mem8*	0F AE /7	Flush cache line containing *mem8*.	✔
CMC	F5	Complement the carry flag (CF).	✔
CMOVO *reg16,reg/mem16*	0F 40 /r	Move if overflow (OF = 1).	✔
CMOVO *reg32,reg/mem32*	0F 40 /r	Move if overflow (OF = 1).	✔
CMOVO *reg64,reg/mem64*	0F 40 /r	Move if overflow (OF = 1).	✔
CMOVNO *reg16,reg/mem16*	0F 41 /r	Move if not overflow (OF = 0).	✔
CMOVNO *reg32,reg/mem32*	0F 41 /r	Move if not overflow (OF = 0).	✔
CMOVNO *reg64,reg/mem64*	0F 41 /r	Move if not overflow (OF = 0).	✔
CMOVB *reg16,reg/mem16*	0F 42 /r	Move if below (CF = 1).	✔
CMOVB *reg32,reg/mem32*	0F 42 /r	Move if below (CF = 1).	✔
CMOVB *reg64,reg/mem64*	0F 42 /r	Move if below (CF = 1).	✔
CMOVC *reg16,reg/mem16*	0F 42 /r	Move if carry (CF = 1).	✔
CMOVC *reg32,reg/mem32*	0F 42 /r	Move if carry (CF = 1).	✔
CMOVC *reg64,reg/mem64*	0F 42 /r	Move if carry (CF = 1).	✔
CMOVNAE *reg16,reg/mem16*	0F 42 /r	Move if not above or equal (CF = 1).	✔
CMOVNAE *reg32,reg/mem32*	0F 42 /r	Move if not above or equal (CF = 1).	✔
CMOVNAE *reg64,reg/mem64*	0F 42 /r	Move if not above or equal (CF = 1).	✔
CMOVNB *reg16,reg/mem16*	0F 43 /r	Move if not below (CF = 0).	✔
CMOVNB *reg32,reg/mem32*	0F 43 /r	Move if not below (CF = 0).	✔
CMOVNB *reg64,reg/mem64*	0F 43 /r	Move if not below (CF = 0).	✔
CMOVNC *reg16,reg/mem16*	0F 43 /r	Move if not carry (CF = 0).	✔
CMOVNC *reg32,reg/mem32*	0F 43 /r	Move if not carry (CF = 0).	✔
CMOVNC *reg64,reg/mem64*	0F 43 /r	Move if not carry (CF = 0).	✔
CMOVAE *reg16,reg/mem16*	0F 43 /r	Move if above or equal (CF = 0).	✔
CMOVAE *reg32,reg/mem32*	0F 43 /r	Move if above or equal (CF = 0).	✔
CMOVAE *reg64,reg/mem64*	0F 43 /r	Move if above or equal (CF = 0).	✔
CMOVZ *reg16,reg/mem16*	0F 44 /r	Move if zero (ZF = 1).	✔
CMOVZ *reg32,reg/mem32*	0F 44 /r	Move if zero (ZF = 1).	✔
CMOVZ *reg64,reg/mem64*	0F 44 /r	Move if zero (ZF = 1).	✔
CMOVE *reg16,reg/mem16*	0F 44 /r	Move if equal (ZF = 1).	✔
CMOVE *reg32,reg/mem32*	0F 44 /r	Move if equal (ZF = 1).	✔
CMOVE *reg64,reg/mem64*	0F 44 /r	Move if equal (ZF = 1).	✔
CMOVNZ *reg16,reg/mem16*	0F 45 /r	Move if not zero (ZF = 0).	✔
CMOVNZ *reg32,reg/mem32*	0F 45 /r	Move if not zero (ZF = 0).	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
CMOVNZ *reg64,reg/mem64*	0F 45 /r	Move if not zero (ZF = 0).	✔
CMOVNE *reg16,reg/mem16*	0F 45 /r	Move if not equal (ZF = 0).	✔
CMOVNE *reg32,reg/mem32*	0F 45 /r	Move if not equal (ZF = 0).	✔
CMOVNE *reg64,reg/mem64*	0F 45 /r	Move if not equal (ZF = 0).	✔
CMOVBE *reg16,reg/mem16*	0F 46 /r	Move if below or equal (CF = 1 or ZF = 1).	✔
CMOVBE *reg32,reg/mem32*	0F 46 /r	Move if below or equal (CF = 1 or ZF = 1).	✔
CMOVBE *reg64,reg/mem64*	0F 46 /r	Move if below or equal (CF = 1 or ZF = 1).	✔
CMOVNA *reg16,reg/mem16*	0F 46 /r	Move if not above (CF = 1 or ZF = 1).	✔
CMOVNA *reg32,reg/mem32*	0F 46 /r	Move if not above (CF = 1 or ZF = 1).	✔
CMOVNA *reg64,reg/mem64*	0F 46 /r	Move if not above (CF = 1 or ZF = 1).	✔
CMOVNBE *reg16,reg/mem16*	0F 47 /r	Move if not below or equal (CF = 0 or ZF = 0).	✔
CMOVNBE *reg32,reg/mem32*	0F 47 /r	Move if not below or equal (CF = 0 or ZF = 0).	✔
CMOVNBE *reg64,reg/mem64*	0F 47 /r	Move if not below or equal (CF = 0 or ZF = 0).	✔
CMOVA *reg16,reg/mem16*	0F 47 /r	Move if above (CF = 1 or ZF = 0).	✔
CMOVA *reg32,reg/mem32*	0F 47 /r	Move if above (CF = 1 or ZF = 0).	✔
CMOVA *reg64,reg/mem64*	0F 47 /r	Move if above (CF = 1 or ZF = 0).	✔
CMOVS *reg16,reg/mem16*	0F 48 /r	Move if sign (SF = 1).	✔
CMOVS *reg32,reg/mem32*	0F 48 /r	Move if sign (SF = 1).	✔
CMOVS *reg64,reg/mem64*	0F 48 /r	Move if sign (SF = 1).	✔
CMOVNS *reg16,reg/mem16*	0F 49 /r	Move if not sign (SF = 0).	✔
CMOVNS *reg32,reg/mem32*	0F 49 /r	Move if not sign (SF = 0).	✔
CMOVNS *reg64,reg/mem64*	0F 49 /r	Move if not sign (SF = 0).	✔
CMOVP *reg16,reg/mem16*	0F 4A /r	Move if parity (PF = 1).	✔
CMOVP *reg32,reg/mem32*	0F 4A /r	Move if parity (PF = 1).	✔
CMOVP *reg64,reg/mem64*	0F 4A /r	Move if parity (PF = 1).	✔
CMOVPE *reg16,reg/mem16*	0F 4A /r	Move if parity even (PF = 1).	✔
CMOVPE *reg32,reg/mem32*	0F 4A /r	Move if parity even (PF = 1).	✔
CMOVPE *reg64,reg/mem64*	0F 4A /r	Move if parity even (PF = 1).	✔
CMOVNP *reg16,reg/mem16*	0F 4B /r	Move if not parity (PF = 0).	✔
CMOVNP *reg32,reg/mem32*	0F 4B /r	Move if not parity (PF = 0).	✔
CMOVNP *reg64,reg/mem64*	0F 4B /r	Move if not parity (PF = 0).	✔
CMOVPO *reg16,reg/mem16*	0F 4B /r	Move if parity odd (PF = 0).	✔
CMOVPO *reg32,reg/mem32*	0F 4B /r	Move if parity odd (PF = 0).	✔
CMOVPO *reg64,reg/mem64*	0F 4B /r	Move if parity odd (PF = 0).	✔
CMOVL *reg16,reg/mem16*	0F 4C /r	Move if less (SF <> OF).	✔
CMOVL *reg32,reg/mem32*	0F 4C /r	Move if less (SF <> OF).	✔
CMOVL *reg64,reg/mem64*	0F 4C /r	Move if less (SF <> OF).	✔
CMOVNGE *reg16,reg/mem16*	0F 4C /r	Move if not greater or equal (SF <> OF).	✔
CMOVNGE *reg32,reg/mem32*	0F 4C /r	Move if not greater or equal (SF <> OF).	✔
CMOVNGE *reg64,reg/mem64*	0F 4C /r	Move if not greater or equal (SF <> OF).	✔
CMOVNL *reg16,reg/mem16*	0F 4D /r	Move if not less (SF = OF).	✔
CMOVNL *reg32,reg/mem32*	0F 4D /r	Move if not less (SF = OF).	✔
CMOVNL *reg64,reg/mem64*	0F 4D /r	Move if not less (SF = OF).	✔
CMOVGE *reg16,reg/mem16*	0F 4D /r	Move if greater or equal (SF = OF).	✔
CMOVGE *reg32,reg/mem32*	0F 4D /r	Move if greater or equal (SF = OF).	✔
CMOVGE *reg64,reg/mem64*	0F 4D /r	Move if greater or equal (SF = OF).	✔
CMOVLE *reg16,reg/mem16*	0F 4E /r	Move if less or equal (ZF = 1 or SF <> OF).	✔
CMOVLE *reg32,reg/mem32*	0F 4E /r	Move if less or equal (ZF = 1 or SF <> OF).	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
CMOVLE *reg64,reg/mem64*	0F 4E /r	Move if less or equal (ZF = 1 or SF <> OF).	✔
CMOVNG *reg16,reg/mem16*	0F 4E /r	Move if less not greater (ZF = 1 or SF <> OF).	✔
CMOVNG *reg32,reg/mem32*	0F 4E /r	Move if less not greater (ZF = 1 or SF <> OF).	✔
CMOVNG *reg64,reg/mem64*	0F 4E /r	Move if less not greater (ZF = 1 or SF <> OF).	✔
CMOVNLE *reg16,reg/mem16*	0F 4F /r	Move if not less or equal (ZF = 0 or SF = OF).	✔
CMOVNLE *reg32,reg/mem32*	0F 4F /r	Move if not less or equal (ZF = 0 or SF = OF).	✔
CMOVNLE *reg64,reg/mem64*	0F 4F /r	Move if not less or equal (ZF = 0 or SF = OF).	✔
CMOVG *reg16,reg/mem16*	0F 4F /r	Move if greater (ZF = 0 or SF = OF).	✔
CMOVG *reg32,reg/mem32*	0F 4F /r	Move if greater (ZF = 0 or SF = OF).	✔
CMOVG *reg64,reg/mem64*	0F 4F /r	Move if greater (ZF = 0 or SF = OF).	✔
CMP AL,*imm8*	3C *ib*	Compare an 8-bit immediate value with the contents of the AL register.	✔
CMP AX,*imm16*	3D *iw*	Compare a 16-bit immediate value with the contents of the AX register.	✔
CMP EAX,*imm32*	3D *id*	Compare a 32-bit immediate value with the contents of the EAX register.	✔
CMP RAX,*imm32*	3D *id*	Compare a 32-bit immediate value with the contents of the RAX register.	✔
CMP *reg/mem8,imm8*	80 /7 *ib*	Compare an 8-bit value with the contents of an 8-bit register or memory operand.	✔
CMP *reg/mem16,imm16*	81 /7 *iw*	Compare a 16-bit value with the contents of a 16-bit register or memory operand.	✔
CMP *reg/mem32,imm32*	81 /7 *id*	Compare a 32-bit value with the contents of a 32-bit register or memory operand.	✔
CMP *reg/mem64,imm32*	81 /7 *id*	Compare a 32-bit signed immediate value with the contents of a 64-bit register or memory operand.	✔
CMP *reg/mem16,imm8*	83 /7 *ib*	Compare an 8-bit signed immediate value with the contents of a 16-bit register or memory operand.	✔
CMP *reg/mem32,imm8*	83 /7 *id*	Compare an 8-bit signed immediate value with the contents of a 32-bit register or memory operand.	✔
CMP *reg/mem64,imm8*	83 /7 *id*	Compare an 8-bit signed immediate value with the contents of a 64-bit register or memory operand.	✔
CMP *reg/mem8,reg8*	38 /r	Compare the contents of an 8-bit register or memory operand with the contents of an 8-bit register.	✔
CMP *reg/mem16,reg16*	39 /r	Compare the contents of a 16-bit register or memory operand with the contents of a 16-bit register.	✔
CMP *reg/mem32,reg32*	39 /r	Compare the contents of a 32-bit register or memory operand with the contents of a 32-bit register.	✔
CMP *reg/mem64,reg64*	39 /r	Compare the contents of a 64-bit register or memory operand with the contents of a 64-bit register.	✔
CMP *reg8,reg/mem8*	3A /r	Compare the contents of an 8-bit register with the contents of an 8-bit register or memory operand.	✔
CMP *reg16,reg/mem16*	3B /r	Compare the contents of a 16-bit register with the contents of a 16-bit register or memory operand.	✔
CMP *reg32,reg/mem32*	3B /r	Compare the contents of a 32-bit register with the contents of a 32-bit register or memory operand.	✔
CMP *reg64,reg/mem64*	3B /r	Compare the contents of a 64-bit register with the contents of a 64-bit register or memory operand.	✔
CMPS *mem8,mem8*	A6	Compare the byte at DS:rSI with the byte at ES:rDI and then increment or decrement rSI and rDI.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
CMPS *mem16,mem16*	A7	Compare the word at DS:rSI with the word at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPS *mem32,mem32*	A7	Compare the doubleword at DS:rSI with the doubleword at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPS *mem64,mem64*	A7	Compare the quadword at DS:rSI with the quadword at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPSB	A6	Compare the byte at DS:rSI with the byte at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPSW	A7	Compare the word at DS:rSI with the word at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPSD	A7	Compare the doubleword at DS:rSI with the doubleword at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPSQ	A7	Compare the quadword at DS:rSI with the quadword at ES:rDI and then increment or decrement rSI and rDI.	✔
CMPXCHG *reg/mem8,reg8*	0F B0 /r	Compare AL register with an 8-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to AL.	✔
CMPXCHG *reg/mem16,reg16*	0F B1 /r	Compare AX register with a 16-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to AX.	✔
CMPXCHG *reg/mem32,reg32*	0F B1 /r	Compare EAX register with a 32-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to EAX.	✔
CMPXCHG *reg/mem64,reg64*	0F B1 /r	Compare RAX register with a 64-bit register or memory location. If equal, copy the second operand to the first operand. Otherwise, copy the first operand to RAX.	✔
CMPXCHG8B	0F C7 /1 *m64*	Compare EDX:EAX register to 64-bit memory location. If equal, set the zero flag (ZF) to 1 and copy the ECX:EBX register to the memory location. Otherwise, copy the memory location to EDX:EAX and clear the zero flag.	✔
CPUID	0F A2	Executes the CPUID function whose number is in the EAX register.	✔
DAA	27	Decimal adjust AL.	✔
DAS	2F	Decimal adjusts AL after subtraction.	✔
DEC *reg/mem8*	FE /1	Decrement the contents of an 8-bit register or memory location by 1.	✔
DEC *reg/mem16*	FF /1	Decrement the contents of a 16-bit register or memory location by 1.	✔
DEC *reg/mem32*	FF /1	Decrement the contents of a 32-bit register or memory location by 1.	✔
DEC *reg/mem64*	FF /1	Decrement the contents of a 64-bit register or memory location by 1.	✔
DEC *reg16*	48 *+rw*	Decrement the contents of a 16-bit register by 1.	✔
DEC *reg32*	48 *+rd*	Decrement the contents of a 32-bit register by 1.	✔
DIV *reg/mem8*	F6 /6	Perform unsigned division of AX by the contents of an 8-bit register or memory location and store the quotient in AL and the remainder in AH.	✔
DIV *reg/mem16*	F7 /6	Perform unsigned division of DX:AX by the contents of a 16-bit register or memory location and store the quotient in AX and the remainder in DX.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
DIV *reg/mem32*	F7 /6	Perform unsigned division of EDX:EAX by the contents of a 32-bit register or memory location and store the quotient in EAX and the remainder in EDX.	✔
DIV *reg/mem64*	F7 /6	Perform unsigned division of RDX:RAX by the contents of a 64-bit register or memory location and store the quotient in RAX and the remainder in RDX.	✔
ENTER *imm16*,0	CB *iw* 00	Create a procedure stack frame.	⚠
ENTER *imm16*,1	CB *iw* 01	Create a nested stack frame for a procedure.	⚠
ENTER *imm16*,*imm8*	CB *iw ib*	Create a nested stack frame for a procedure.	⚠
IDIV *reg/mem8*	F6 /7	Perform signed division of AX by the contents of an 8-bit register or memory location and store the quotient in AL and the remainder in AH.	✔
IDIV *reg/mem16*	F7 /7	Perform signed division of DX:AX by the contents of a 16-bit register or memory location and store the quotient in AX and the remainder in DX.	✔
IDIV *reg/mem32*	F7 /7	Perform signed division of EDX:EAX by the contents of a 32-bit register or memory location and store the quotient in EAX and the remainder in EDX.	✔
IDIV *reg/mem64*	F7 /7	Perform signed division of RDX:RAX by the contents of a 64-bit register or memory location and store the quotient in RAX and the remainder in RDX.	✔
IMUL *reg/mem8*	F6 /5	Multiply the contents of AL by the contents of an 8-bit memory or register operand and put the signed result in AX.	✔
IMUL *reg/mem16*	F7 /5	Multiply the contents of AX by the contents of a 16-bit memory or register operand and put the signed result in DX:AX.	✔
IMUL *reg/mem32*	F7 /5	Multiply the contents of EAX by the contents of a 32-bit memory or register operand and put the signed result in EDX:EAX.	✔
IMUL *reg/mem64*	F7 /5	Multiply the contents of RAX by the contents of a 64-bit memory or register operand and put the signed result in RDX:RAX.	✔
IMUL *reg16*,*reg/mem16*	OF AF /r	Multiply the contents of a 16-bit destination register by the contents of a 16-bit register or memory operand and put the signed result the 16-bit destination register.	✔
IMUL *reg32*,*reg/mem32*	OF AF /r	Multiply the contents of a 32-bit destination register by the contents of a 32-bit register or memory operand and put the signed result the 32-bit destination register.	✔
IMUL *reg64*,*reg/mem64*	OF AF /r	Multiply the contents of a 64-bit destination register by the contents of a 64-bit register or memory operand and put the signed result the 64-bit destination register.	✔
IMUL *reg16*,*reg/mem16*,*imm8*	6B /r *ib*	Multiply the contents of a 16-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 16-bit destination register.	✔
IMUL *reg32*,*reg/mem32*,*imm8*	6B /r *ib*	Multiply the contents of a 32-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 32-bit destination register.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
IMUL *reg64,reg/mem64,imm8*	6B /r *ib*	Multiply the contents of a 64-bit register or memory operand by a sign-extended immediate byte and put the signed result in the 64-bit destination register.	✔
IMUL *reg16,reg/mem16,imm16*	69 /r *iw*	Multiply the contents of a 16-bit register or memory operand by a sign-extended immediate word and put the signed result in the 16-bit destination register.	✔
IMUL *reg32,reg/mem32,imm32*	69 /r *id*	Multiply the contents of a 32-bit register or memory operand by a sign-extended immediate double and put the signed result in the 32-bit destination register.	✔
IMUL *reg64,reg/mem64,imm32*	69 /r *id*	Multiply the contents of a 64-bit register or memory operand by a sign-extended immediate double and put the signed result in the 64-bit destination register.	✔
IN AL,*imm8*	E4 *ib*	Input a byte from the port at the address specified by *imm8* and put it into the AL register.	✔
IN AX,*imm8*	E5 *ib*	Input a word from the port at the address specified by *imm8* and put it into the AX register.	✔
IN EAX,*imm8*	E5 *ib*	Input a doubleword from the port at the address specified by *imm8* and put it into the EAX register.	✔
IN AL,DX	EC	Input a byte from the port at the address specified by the DX register and put it into the AL register.	✔
IN AX,DX	ED	Input a word from the port at the address specified by the DX register and put it into the AX register.	✔
IN EAX,EDX	ED	Input a doubleword from the port at the address specified by the EDX register and put it into the EAX register.	✔
INC *reg/mem8*	FE /0	Increment the contents of an 8-bit register or memory location by 1.	✔
INC *reg/mem16*	FF /0	Increment the contents of a 16-bit register or memory location by 1.	✔
INC *reg/mem32*	FF /0	Increment the contents of a 32-bit register or memory location by 1.	✔
INC *reg/mem64*	FF /0	Increment the contents of a 64-bit register or memory location by 1.	✔
INC *reg16*	40 *+rw*	Increment the contents of a 16-bit register by 1.	✔
INC *reg32*	40 *+rd*	Increment the contents of a 32-bit register by 1.	✔
INS *mem8*,DX	6C	Input a byte from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.	✔
INS *mem16*,DX	6D	Input a word from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.	✔
INS *mem32*,DX	6D	Input a doubleword from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.	✔
INSB	6C	Input a byte from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.	✔
INSW	6D	Input a word from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.	✔
INSD	6D	Input a doubleword from the port specified by DX, put it into the memory location specified in ES:rDI, and then increment or decrement rDI.	✔
INT *imm8*	CD *ib*	Calls interrupt service routine specified by interrupt vector *imm8*.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
INTO	CE	Calls overflow exception if the overflow flag is set.	✔
JO *rel8off*	80 *cb*	Jump if overflow (OF = 1).	✔
JO *rel16off*	0F 80 *cw*	Jump if overflow (OF = 1).	✔
JO *rel32off*	0F 80 *cd*	Jump if overflow (OF = 1).	✔
JNO *rel8off*	71 *cb*	Jump if not overflow (OF = 0)	✔
JNO *rel16off*	0F 81 *cw*	Jump if not overflow (OF = 0)	✔
JNO *rel32off*	0F 81 *cd*	Jump if not overflow (OF = 0)	✔
JB *rel8off*	72 *cb*	Jump if below (CF = 1).	✔
JB *rel16off*	0F 82 *cw*	Jump if below (CF = 1).	✔
JB *rel32off*	0F 82 *cd*	Jump if below (CF = 1).	✔
JC *rel8off*	72 *cb*	Jump if carry (CF =1).	✔
JC *rel16off*	0F 82 *cw*	Jump if carry (CF =1).	✔
JC *rel32off*	0F 82 *cd*	Jump if carry (CF =1).	✔
JNAE *rel8off*	72 *cb*	Jump if not above or equal (CF =1).	✔
JNAE *rel16off*	0F 82 *cw*	Jump if not above or equal (CF =1).	✔
JNAE *rel32off*	0F 82 *cd*	Jump if not above or equal (CF =1).	✔
JNB *rel8off*	73 *cb*	Jump if not below (CF = 0).	✔
JNB *rel16off*	0F 83 *cw*	Jump if not below (CF = 0).	✔
JNB *rel32off*	0F 83 *cd*	Jump if not below (CF = 0).	✔
JNC *rel8off*	73 *cb*	Jump if not carry (CF = 0).	✔
JNC *rel16off*	0F 83 *cw*	Jump if not carry (CF = 0).	✔
JNC *rel32off*	0F 83 *cd*	Jump if not carry (CF = 0).	✔
JAE *rel8off*	73 *cb*	Jump if above or equal (CF = 0).	✔
JAE *rel16off*	0F 83 *cw*	Jump if above or equal (CF = 0).	✔
JAE *rel32off*	0F 83 *cd*	Jump if above or equal (CF = 0).	✔
JZ *rel8off*	74 *cb*	Jump if zero (ZF =1).	✔
JZ *rel16off*	0F 84 *cw*	Jump if zero (ZF =1).	✔
JZ *rel32off*	0F 84 *cd*	Jump if zero (ZF =1).	✔
JE *rel8off*	74 *cb*	Jump if equal (ZF =1).	✔
JE *rel16off*	0F 84 *cw*	Jump if equal (ZF =1).	✔
JE *rel32off*	0F 84 *cd*	Jump if equal (ZF =1).	✔
JNZ *rel8off*	75 *cb*	Jump if not zero (ZF = 0).	✔
JNZ *rel16off*	0F 85 *cw*	Jump if not zero (ZF = 0).	✔
JNZ *rel32off*	0F 85 *cd*	Jump if not zero (ZF = 0).	✔
JNE *rel8off*	75 *cb*	Jump if not equal (ZF = 0).	✔
JNE *rel16off*	0F 85 *cw*	Jump if not equal (ZF = 0).	✔
JNE *rel32off*	0F 85 *cd*	Jump if not equal (ZF = 0).	✔
JBE *rel8off*	76 *cb*	Jump if below or equal (CF = 1 or ZF = 1).	✔
JBE *rel16off*	0F 86 *cw*	Jump if below or equal (CF = 1 or ZF = 1).	✔
JBE *rel32off*	0F 86 *cd*	Jump if below or equal (CF = 1 or ZF = 1).	✔
JNA *rel8off*	76 *cb*	Jump if not above (CF = 1 or ZF = 1).	✔
JNA *rel16off*	0F 86 *cw*	Jump if not above (CF = 1 or ZF = 1).	✔
JNA *rel32off*	0F 86 *cd*	Jump if not above (CF = 1 or ZF = 1).	✔
JNBE *rel8off*	77 *cb*	Jump if not below or equal (CF = 0 or ZF = 0).	✔
JNBE *rel16off*	0F 87 *cw*	Jump if not below or equal (CF = 0 or ZF = 0).	✔
JNBE *rel32off*	0F 87 *cd*	Jump if not below or equal (CF = 0 or ZF = 0).	✔
JA *rel8off*	77 *cb*	Jump if above (CF = 0 or ZF = 0).	✔
JA *rel16off*	0F 87 *cw*	Jump if above (CF = 0 or ZF = 0).	✔
JA *rel32off*	0F 87 *cd*	Jump if above (CF = 0 or ZF = 0).	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
JS *rel8off*	78 *cb*	Jump if sign (SF = 1).	✔
JS *rel16off*	0F 88 *cw*	Jump if sign (SF = 1).	✔
JS *rel32off*	0F 88 *cd*	Jump if sign (SF = 1).	✔
JNS *rel8off*	79 *cb*	Jump if not sign (SF = 0).	✔
JNS *rel16off*	0F 89 *cw*	Jump if not sign (SF = 0).	✔
JNS *rel32off*	0F 89 *cd*	Jump if not sign (SF = 0).	✔
JP *rel8off*	7A *cb*	Jump if parity (PF = 1).	✔
JP *rel16off*	0F 8A *cw*	Jump if parity (PF = 1).	✔
JP *rel32off*	0F 8A *cd*	Jump if parity (PF = 1).	✔
JPE *rel8off*	7A *cb*	Jump if parity even (PF = 1).	✔
JPE *rel16off*	0F 8A *cw*	Jump if parity even (PF = 1).	✔
JPE *rel32off*	0F 8A *cd*	Jump if parity even (PF = 1).	✔
JNP *rel8off*	7B *cb*	Jump if not parity (PF = 0).	✔
JNP *rel16off*	0F 8B *cw*	Jump if not parity (PF = 0).	✔
JNP *rel32off*	0F 8B *cd*	Jump if not parity (PF = 0).	✔
JPO *rel8off*	7B *cb*	Jump if parity odd (PF = 0).	✔
JPO *rel16off*	0F 8B *cw*	Jump if parity odd (PF = 0).	✔
JPO *rel32off*	0F 8B *cd*	Jump if parity odd (PF = 0).	✔
JL *rel8off*	7C *cb*	Jump if less (SF <> OF).	✔
JL *rel16off*	0F 8C *cw*	Jump if less (SF <> OF).	✔
JL *rel32off*	0F 8C *cd*	Jump if less (SF <> OF).	✔
JNGE *rel8off*	7C *cb*	Jump if not greater or equal (SF <> OF).	✔
JNGE *rel16off*	0F 8C *cw*	Jump if not greater or equal (SF <> OF).	✔
JNGE *rel32off*	0F 8C *cd*	Jump if not greater or equal (SF <> OF).	✔
JNL *rel8off*	7D *cb*	Jump if not less (SF = OF).	✔
JNL *rel16off*	0F 8D *cw*	Jump if not less (SF = OF).	✔
JNL *rel32off*	0F 8D *cd*	Jump if not less (SF = OF).	✔
JGE *rel8off*	7D *cb*	Jump if greater or equal (SF = OF).	✔
JGE *rel16off*	0F 8D *cw*	Jump if greater or equal (SF = OF).	✔
JGE *rel32off*	0F 8D *cd*	Jump if greater or equal (SF = OF).	✔
JLE *rel8off*	7E *cb*	Jump if less or equal (ZF = 1 or SF <> OF).	✔
JLE *rel16off*	0F 8R *cw*	Jump if less or equal (ZF = 1 or SF <> OF).	✔
JLE *rel32off*	0F 8R *cd*	Jump if less or equal (ZF = 1 or SF <> OF).	✔
JNG *rel8off*	7E *cb*	Jump if not greater (ZF = 1 or SF <> OF).	✔
JNG *rel16off*	0F 8E *cw*	Jump if not greater (ZF = 1 or SF <> OF).	✔
JNG *rel32off*	0F 8E *cd*	Jump if not greater (ZF = 1 or SF <> OF).	✔
JNLE *rel8off*	7F *cb*	Jump if not less or equal (ZF = 0 or SF = OF).	✔
JNLE *rel16off*	0F 8F *cw*	Jump if not less or equal (ZF = 0 or SF = OF).	✔
JNLE *rel32off*	0F 8F *cd*	Jump if not less or equal (ZF = 0 or SF = OF).	✔
JG *rel8off*	7F *cb*	Jump if greater (ZF = 0 or SF = OF).	✔
JG *rel16off*	0F 8F *cw*	Jump if greater (ZF = 0 or SF = OF).	✔
JG *rel32off*	0F 8F *cd*	Jump if greater (ZF = 0 or SF = OF).	✔
JCXZ *rel8off*	E3 *cb*	Jump short if the 16-bit count register (CX) is zero.	✔
JCXZ *rel16off*	E3 *cb*	Jump short if the 32-bit count register (ECX) is zero.	✔
JCXZ *rel32off*	E3 *cb*	Jump short if the 32-bit count register (RCX) is zero.	✔
JMP *rel8off*	EB *cb*	Short jump with the target specified by an 8-bit signed displacement.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
JMP *rel16off*	E9 *cw*	Short jump with the target specified by a 16-bit signed displacement.	✔
JMP *rel32off*	E9 *cd*	Short jump with the target specified by a 32-bit signed displacement.	✔
JMP *reg/mem16*	FF /4	Near jump with the target specified *reg/mem16*.	✔
JMP *reg/mem32*	FF /4	Near jump with the target specified *reg/mem32*.	✔
JMP *reg/mem64*	FF /4	Near jump with the target specified *reg/mem64*.	✔
JMP FAR *pntr16:16*	EA *cd*	Far jump direct, with the target specified by a far pointer contained in the instruction.	✔
JMP FAR *pntr16:32*	EA *cp*	Far jump direct, with the target specified by a far pointer contained in the instruction.	✔
JMP FAR *mem16:16*	FF /5	Far jump indirect, with the target specified by a far pointer in memory.	✔
JMP FAR *mem16:32*	FF /5	Far jump indirect, with the target specified by a far pointer in memory.	✔
LAHF	9F	Load the SF, ZF, AF, PF, and CF flags into the AH register.	✔
LDS *reg16,mem16:16*	C5 /r	Load DS:reg16 with a far pointer from memory.	✔
LDS *reg32,mem16:32*	C5 /r	Load DS:reg32 with a far pointer from memory.	✔
LES *reg16,mem16:16*	C4 /r	Load ES:reg16 with a far pointer from memory.	✔
LES *reg32,mem16:32*	C4 /r	Load ES:reg32 with a far pointer from memory.	✔
LFS *reg16,mem16:16*	0F B4 /r	Load FS:reg16 with a far pointer from memory.	✔
LFS *reg32,mem16:32*	0F B4 /r	Load FS:reg32 with a far pointer from memory.	✔
LGS *reg16,mem16:16*	0F B5 /r	Load GS:reg16 with a far pointer from memory.	✔
LGS *reg32,mem16:32*	0F B5 /r	Load GS:reg32 with a far pointer from memory.	✔
LSS *reg16,mem16:16*	0F B2 /r	Load SS:reg16 with a far pointer from memory.	✔
LSS *reg32,mem16:32*	0F B2 /r	Load SS:reg32 with a far pointer from memory.	✔
LEA *reg16,mem*	8D /r	Store effective address in a 16-bit register.	✔
LEA *reg32,mem*	8D /r	Store effective address in a 32-bit register.	✔
LEA *reg64,mem*	8D /r	Store effective address in a 64-bit register.	✔
LEAVE	C9	Set the stack pointer SP to the value in the BP register and pop BP.	⚠
LEAVE	C9	Set the stack pointer ESP to the value in the EBP register and pop EBP.	⚠
LEAVE	C9	Set the stack pointer RSP to the value in the RBP register and pop RBP.	⚠
LFENCE	0F AE E8	Force strong ordering of (serialize) load operations.	✔
LODS *mem8*	AC	Load byte at DS:rSI into AL and then increment or decrement rSI.	✔
LODS *mem16*	AD	Load word at DS:rSI into AX and then increment or decrement rSI.	✔
LODS *mem32*	AD	Load doubleword at DS:rSI into EAX and then increment or decrement rSI.	✔
LODS *mem64*	AD	Load quadword at DS:rSI into RAX and then increment or decrement rSI.	✔
LODSB	AC	Load byte at DS:rSI into AL and then increment or decrement rSI.	✔
LODSW	AD	Load word at DS:rSI into AX and then increment or decrement rSI.	✔
LODSD	AD	Load doubleword at DS:rSI into EAX and then increment or decrement rSI.	✔
LODSQ	AD	Load quadword at DS:rSI into RAX and then increment or decrement rSI.	✔
LOOP *rel8off*	E2 *cb*	Decrement rCX and then jump short if rCX is not 0.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
LOOPE *rel8off*	E1 *cb*	Decrement rCX and then jump short if rCX is not 0 and ZF is 1.	✔
LOOPNE *rel8off*	E0 *cb*	Decrement rCX and then jump short if rCX is not 0 and ZF is 0.	✔
LOOPNZ *rel8off*	E0 *cb*	Decrement rCX and then jump short if rCX is not 0 and ZF is 0.	✔
LOOPZ *rel8off*	E1 *cb*	Decrement rCX and then jump short if rCX is not 0 and ZF is 1.	✔
MFENCE	0F AE F0	Force strong ordering of (serialized) load and store operations.	✔
MOV *reg/mem8,reg8*	88 /r	Move the contents of an 8-bit register to an 8-bit destination register or memory operand.	✔
MOV *reg/mem16,reg16*	89 /r	Move the contents of a 16-bit register to a 16-bit destination register or memory operand.	✔
MOV *reg/mem32,reg32*	89 /r	Move the contents of a 32-bit register to a 32-bit destination register or memory operand.	✔
MOV *reg/mem64,reg64*	89 /r	Move the contents of a 64-bit register to a 64-bit destination register or memory operand.	✔
MOV *reg8,reg/mem8*	8A /r	Move the contents of an 8-bit register or memory operand to an 8-bit destination register.	✔
MOV *reg16,reg/mem16*	8B /r	Move the contents of a 16-bit register or memory operand to a 16-bit destination register.	✔
MOV *reg32,reg/mem32*	8B /r	Move the contents of a 32-bit register or memory operand to a 32-bit destination register.	✔
MOV *reg64,reg/mem64*	8B /r	Move the contents of a 64-bit register or memory operand to a 64-bit destination register.	✔
MOV *reg16/32/64/mem16,segReg*	8C /r	Move the contents of a segment register to a 16-bit, 32-bit, or 64-bit destination register or to a 16-bit memory operand.	✔
MOV *segReg,reg/mem16*	8E /r	Move the contents of a 16-bit register or memory operand to a segment register.	✔
MOV AL,*moffset8*	A0	Move 8-bit data at a specified memory offset to the AL register.	✔
MOV AX,*moffset16*	A1	Move 16-bit data at a specified memory offset to the AX register.	✔
MOV EAX,*moffset32*	A1	Move 32-bit data at a specified memory offset to the EAX register.	✔
MOV RAX,*moffset64*	A1	Move 64-bit data at a specified memory offset to the RAX register.	✔
MOV *moffset8*,AL	A2	Move the contents of the AL register to an 8-bit memory offset.	✔
MOV *moffset16*,AX	A3	Move the contents of the AX register to a 16-bit memory offset.	✔
MOV *moffset32*,EAX	A3	Move the contents of the EAX register to a 32-bit memory offset.	✔
MOV *moffset64*,RAX	A3	Move the contents of the RAX register to a 64-bit memory offset.	✔
MOV *reg8,imm8*	B0 *+rb*	Move an 8-bit immediate value into an 8-bit register.	✔
MOV *reg16,imm16*	B8 *+rw*	Move a 16-bit immediate value into a 16-bit register.	✔
MOV *reg32,imm32*	B8 *+rd*	Move a 32-bit immediate value into a 32-bit register.	✔
MOV *reg64,imm64*	B8 *+rq*	Move a 64-bit immediate value into a 64-bit register.	✔
MOV *reg/mem8,imm8*	C6 /0	Move an 8-bit immediate value to an 8-bit register or memory operand.	✔
MOV *reg/mem16,imm16*	C7 /0	Move a 16-bit immediate value to a 16-bit register or memory operand.	✔
MOV *reg/mem32,imm32*	C7 /0	Move a 32-bit immediate value to a 32-bit register or memory operand.	✔
MOV *reg/mem64,imm64*	C7 /0	Move a 64-bit immediate value to a 64-bit register or memory operand.	✔
MOVD *xmm,reg/mem32*	66 0F 6E /r	Move 32-bit value from a general-purpose register or 32-bit memory location to an XMM register.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
MOVD *xmm,reg/mem64*	66 0F 6E /r	Move 64-bit value from a general-purpose register or 64-bit memory location to an XMM register.	✔
MOVD *reg/mem32,xmm*	66 0F 7E /r	Move 32-bit value from an XMM register to a 32-bit general-purpose register or memory location.	✔
MOVD *reg/mem64,xmm*	66 0F 7E /r	Move 64-bit value from an XMM register to a 64-bit general-purpose register or memory location.	✔
MOVD *mmx,reg/mem32*	0F 6E /r	Move 32-bit value from a general-purpose register or 32-bit memory location to an MMX register.	✔
MOVD *mmx,reg/mem64*	0F 6E /r	Move 64-bit value from a general-purpose register or 64-bit memory location to an MMX register.	✔
MOVD *reg/mem32,mmx*	0F 7E /r	Move 32-bit value from an MMX register to a 32-bit general-purpose register or memory location.	✔
MOVD *reg/mem64,mmx*	0F 7E /r	Move 64-bit value from an MMX register to a 64-bit general-purpose register or memory location.	✔
MOVMSKPD *reg32,xmm*	66 0F 50 /r	Move sign bits 127 and 63 in an XMM register t0 a 32-bit general purpose-register.	✔
MOVMSKPS *reg32,xmm*	0F 50 /r	Move sign bits 127, 95, 63, 31 in an XMM register to a 32-bit general-purpose register.	✔
MOVNTI *mem32,reg32*	0F C3 /r	Stores a 32-bit general-purpose register value into a 32-bit memory location, minimizing cache pollution.	✔
MOVNTI *mem64,reg64*	0F C3 /r	Stores a 64-bit general-purpose register value into a 64-bit memory location, minimizing cache pollution.	✔
MOVS *mem8,mem8*	A4	Move byte at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVS *mem16,mem16*	A5	Move word at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVS *mem32,mem32*	A5	Move doubleword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVS *mem64,mem64*	A5	Move quadword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVSB	A4	Move byte at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVSW	A5	Move word at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVSD	A5	Move doubleword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVSQ	A5	Move quadword at DS:rSI to ES:rDI, and then increment or decrement rSI and rDI.	✔
MOVSX *reg16,reg/mem8*	0F BE /r	Move the contents of an 8-bit register or memory location to a 16-bit register with sign extension.	✔
MOVSX *reg32,reg/mem8*	0F BE /r	Move the contents of an 8-bit register or memory location to a 32-bit register with sign extension.	✔
MOVSX *reg64,reg/mem8*	0F BE /r	Move the contents of an 8-bit register or memory location to a 64-bit register with sign extension.	✔
MOVSX *reg32,reg/mem16*	0F BF /r	Move the contents of a 16-bit register or memory location to a 32-bit register with sign extension.	✔
MOVSX *reg64,reg/mem16*	0F BF /r	Move the contents of a 16-bit register or memory location to a 64-bit register with sign extension.	✔
MOVSXD *reg64,reg/mem32*	63 /r	Move the contents of a 32-bit register or memory operand to a 64-bit register with sign extension.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
MOVZX *reg16,reg/mem8*	0F B6 /r	Move the contents of an 8-bit register or memory operand to a 16-bit register with zero-extension.	✔
MOVZX *reg32,reg/mem8*	0F B6 /r	Move the contents of an 8-bit register or memory operand to a 32-bit register with zero-extension.	✔
MOVZX *reg64,reg/mem8*	0F B6 /r	Move the contents of an 8-bit register or memory operand to a 64-bit register with zero-extension.	✔
MOVZX *reg32,reg/mem16*	0F B7 /r	Move the contents of a 16-bit register or memory operand to a 32-bit register with zero-extension.	✔
MOVZX *reg64,reg/mem16*	0F B7 /r	Move the contents of a 16-bit register or memory operand to a 64-bit register with zero-extension.	✔
MUL *reg/mem8*	F6 /4	Multiplies an 8-bit register or memory operand by the contents of the AL register and stores the result in the AX register.	✔
MUL *reg/mem16*	F7 /4	Multiplies a 16-bit register or memory operand by the contents of the AX register and stores the result in the DX:AX register.	✔
MUL *reg/mem32*	F7 /4	Multiplies a 32-bit register or memory operand by the contents of the EAX register and stores the result in the EDX:EAX register.	✔
MUL *reg/mem64*	F7 /4	Multiplies a 64-bit register or memory operand by the contents of the RAX register and stores the result in the RDX:RAX register.	✔
NEG *reg/mem8*	F6 /3	Performs a tow's complement negation on an 8-bit register or memory operand.	✔
NEG *reg/mem16*	F7 /3	Performs a tow's complement negation on a 16-bit register or memory operand.	✔
NEG *reg/mem32*	F7 /3	Performs a tow's complement negation on a 32-bit register or memory operand.	✔
NEG *reg/mem64*	F7 /3	Performs a tow's complement negation on a 64-bit register or memory operand.	✔
NOP	90	Performs no operation.	✔
NOT *reg/mem8*	F6 /2	Complements the bits in an 8-bit register or memory operand.	✔
NOT *reg/mem16*	F7 /2	Complements the bits in a 16-bit register or memory operand.	✔
NOT *reg/mem32*	F7 /2	Complements the bits in a 32-bit register or memory operand.	✔
NOT *reg/mem64*	F7 /2	Complements the bits in a 64-bit register or memory operand.	✔
OR AL,*imm8*	0C *ib*	OR the contents of AL with an immediate 8-bit value.	✔
OR AX,*imm16*	0D *iw*	OR the contents of AX with an immediate 16-bit value.	✔
OR EAX,*imm32*	0D *id*	OR the contents of EAX with an immediate 32-bit value.	✔
OR RAX,*imm64*	0D *id*	OR the contents of RAX with an immediate 64-bit value.	✔
OR *reg/mem8,imm8*	80 /1 *ib*	OR the contents of an 8-bit register or memory operand and an immediate 8-bit value.	✔
OR *reg/mem16,imm16*	81 /1 *iw*	OR the contents of a 16-bit register or memory operand and an immediate 16-bit value.	✔
OR *reg/mem32,imm32*	81 /1 *id*	OR the contents of a 32-bit register or memory operand and an immediate 32-bit value.	✔
OR *reg/mem64,imm32*	81 /1 *id*	OR the contents of a 64-bit register or memory operand and a sign-extended immediate 32-bit value.	✔
OR *reg/mem16,imm8*	83 /1 *ib*	OR the contents of a 16-bit register or memory operand and a sign-extended immediate 8-bit value.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
OR *reg/mem32,imm8*	83 /1 *ib*	OR the contents of a 32-bit register or memory operand and a sign-extended immediate 8-bit value.	✔
OR *reg/mem64,imm8*	83 /1 *ib*	OR the contents of a 64-bit register or memory operand and a sign-extended immediate 8-bit value.	✔
OR *reg/mem8,reg8*	08 /r	OR the contents of an 8-bit register or memory operand with the contents of an 8-bit register.	✔
OR *reg/mem16,reg16*	09 /r	OR the contents of a 16-bit register or memory operand with the contents of a 16-bit register.	✔
OR *reg/mem32,reg32*	09 /r	OR the contents of a 32-bit register or memory operand with the contents of a 32-bit register.	✔
OR *reg/mem64,reg64*	09 /r	OR the contents of a 64-bit register or memory operand with the contents of a 64-bit register.	✔
OR *reg8,reg/mem8*	0A /r	OR the contents of an 8-bit register with the contents of an 8-bit register or memory operand.	✔
OR *reg16,reg/mem16*	0B /r	OR the contents of a 16-bit register with the contents of a 16-bit register or memory operand.	✔
OR *reg32,reg/mem32*	0B /r	OR the contents of a 32-bit register with the contents of a 32-bit register or memory operand.	✔
OR *reg64,reg/mem64*	0B /r	OR the contents of a 64-bit register with the contents of a 64-bit register or memory operand.	✔
OUT *imm8*,AL	E6 *ib*	Output the byte in the AL register to the port specified by an 8-bit immediate value.	✔
OUT *imm8*,AX	E7 *ib*	Output the word in the AX register to the port specified by an 8-bit immediate value.	✔
OUT *imm8*,EAX	E7 *ib*	Output the doubleword in the EAX register to the port specified by an 8-bit immediate value.	✔
OUT DX,AL	EE	Output the byte in the AL register to the output port specified in DX.	✔
OUT DX,AX	EE	Output the word in the AX register to the output port specified in DX.	✔
OUT DX,EAX	EE	Output the doubleword in the EAX register to the output port specified in DX.	✔
OUTS DX,*mem8*	6E	Output the byte in DS:rSI to the port specified in DX, and then increment or decrement rSI.	✔
OUTS DX,*mem16*	6F	Output the word in DS:rSI to the port specified in DX, and then increment or decrement rSI.	✔
OUTS DX,*mem32*	6F	Output the doubleword in DS:rSI to the port specified in DX, and then increment or decrement rSI.	✔
OUTSB	6E	Output the byte in DS:rSI to the port specified in DX, and then increment or decrement rSI.	✔
OUTSW	6F	Output the word in DS:rSI to the port specified in DX, and then increment or decrement rSI.	✔
OUTSD	6F	Output the doubleword in DS:rSI to the port specified in DX, and then increment or decrement rSI.	✔
POP *reg/mem16*	8F /0	Pop the top of the stack into a 16-bit register or memory location.	✔
POP *reg/mem32*	8F /0	Pop the top of the stack into a 32-bit register or memory location.	✔
POP *reg/mem64*	8F /0	Pop the top of the stack into a 64-bit register or memory location.	✔
POP *reg16*	58 *+rw*	Pop the top of the stack into a 16-bit register.	✔
POP *reg32*	58 *+rd*	Pop the top of the stack into a 32-bit register.	✔
POP *reg64*	58 *+rq*	Pop the top of the stack into a 64-bit register.	

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
POP DS	1F	Pop the top of the stack into the DS register.	✔
POP ES	07	Pop the top of the stack into the ES register.	✔
POP SS	17	Pop the top of the stack into the SS register.	✔
POP FS	0F A1	Pop the top of the stack into the FS register.	✔
POP GS	0F A9	Pop the top of the stack into the GS register.	✔
POPA	61	Pop the DI, SI, BP, SP, BX, DX, CX, and AX registers.	✔
POPAD	61	Pop the EDI, ESI, EBP, ESP, EBX, EDX, ECX, and EAX registers.	✔
POPF	9D	Pop a word from the stack into the FLAGS register.	✔
POPFD	9D	Pop a doubleword from the stack into the EFLAGS register.	✔
POPFQ	9D	Pop a quadword from the stack into the RFLAGS register.	✔
PREFETCH *mem8*	0F 0D /0	Prefetch processor cache line into L1 data cache.	✔
PREFETCHW *mem8*	0F 0D /1	Prefetch processor cache line into L1 data cache and mark it modified.	✔
PREFETCHNTA *mem8*	0F 18 /0	Move data closer to the processor using the NTA reference.	✔
PREFETCHT0 *mem8*	0F 18 /1	Move data closer to the processor using the T0 reference.	✔
PREFETCHT1 *mem8*	0F 18 /2	Move data closer to the processor using the T1 reference.	✔
PREFETCHT2 *mem8*	0F 18 /3	Move data closer to the processor using the T2 reference.	✔
PUSH *reg/mem16*	FF /6	Push the contents of a 16-bit register or memory operand onto the stack.	✔
PUSH *reg/mem32*	FF /6	Push the contents of a 32-bit register or memory operand onto the stack.	✔
PUSH *reg/mem64*	FF /6	Push the contents of a 64-bit register or memory operand onto the stack.	✔
PUSH *reg16*	50 *+rw*	Push the contents of a 16-bit register onto the stack.	✔
PUSH *reg32*	50 *+rd*	Push the contents of a 32-bit register onto the stack.	✔
PUSH *reg64*	50 *+rq*	Push the contents of a 64-bit register onto the stack.	✔
PUSH *imm8*	6A	Push an 8-bit immediate value (sign-extended to 16, 32, or 64 bits) onto the stack.	✔
PUSH *imm16*	68	Push a 16-=bit immediate value onto the stack.	✔
PUSH *imm32*	68	Push the contents of a 32-bit register onto the stack.	✔
PUSH *imm64*	68	Push the contents of a 64-bit register onto the stack.	✔
PUSH CS	0E	Push the CS selector onto the stack.	✔
PUSH SS	16	Push the SS selector onto the stack.	✔
PUSH DS	1E	Push the DS selector onto the stack.	✔
PUSH ES	06	Push the ES selector onto the stack.	✔
PUSH FS	0F A0	Push the FS selector onto the stack.	✔
PUSH GS	0F A8	Push the GS selector onto the stack.	✔
PUSHF	9C	Push the FLAGS word onto the stack.	✔
PUSHFD	9C	Push the EFLAGS word onto the stack.	✔
PUSHFQ	9C	Push the RFLAGS word onto the stack.	✔
RCL *reg/mem8*,1	D0 /2	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left 1 bit.	✔
RCL *reg/mem8*,CL	D2 /2	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified in the CL register.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
RCL *reg/mem8*,*imm8*	C0 /2 *ib*	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
RCL *reg/mem16*,1	D1 /2	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left 1 bit.	✔
RCL *reg/mem16*,CL	D3 /2	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified in the CL register.	✔
RCL *reg/mem16*,*imm8*	C1 /2 *ib*	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
RCL *reg/mem32*,1	D1 /2	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left 1 bit.	✔
RCL *reg/mem32*,CL	D3 /2	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified in the CL register.	✔
RCL *reg/mem32*,*imm8*	C1 /2 *ib*	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
RCL *reg/mem64*,1	D1 /2	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left 1 bit.	✔
RCL *reg/mem64*,CL	D3 /2	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified in the CL register.	✔
RCL *reg/mem64*,*imm8*	C1 /2 *ib*	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
RCR *reg/mem8*,1	D0 /3	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right 1 bit.	✔
RCR *reg/mem8*,CL	D2 /3	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified in the CL register.	✔
RCR *reg/mem8*,*imm8*	C0 /3 *ib*	Rotate the 9 bits consisting of the carry flag and an 8-bit register or memory location right the number of bits specified by an 8-bit immediate value.	✔
RCR *reg/mem16*,1	D1 /3	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right 1 bit.	✔
RCR *reg/mem16*,CL	D3 /3	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified in the CL register.	✔
RCR *reg/mem16*,*imm8*	C1 /3 *ib*	Rotate the 17 bits consisting of the carry flag and a 16-bit register or memory location right the number of bits specified by an 8-bit immediate value.	✔
RCR *reg/mem32*,1	D1 /3	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right 1 bit.	✔
RCR *reg/mem32*,CL	D3 /3	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified in the CL register.	✔
RCR *reg/mem32*,*imm8*	C1 /3 *ib*	Rotate the 33 bits consisting of the carry flag and a 32-bit register or memory location right the number of bits specified by an 8-bit immediate value.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
RCL *reg/mem64*,1 RCR	D1 /3	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right 1 bit.	✔
RCR *reg/mem64*,CL	D3 /3	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified in the CL register.	✔
RCR *reg/mem64*,*imm8*	C1 /3 *ib*	Rotate the 65 bits consisting of the carry flag and a 64-bit register or memory location right the number of bits specified by an 8-bit immediate value.	✔
RET	C3	Near return to the calling procedure.	✔
RET *imm16*	C2 *iw*	Near return to the calling procedure and then pop of the specified number of bytes from the stack.	✔
RETF	CB	Far return to the calling procedure.	✔
RETF *imm16*	CA *iw*	Far return to the calling procedure and then pop of the specified number of bytes from the stack.	✔
ROL *reg/imm8*,1	D0 /0	Rotate an 8-bit register or memory operand left 1 bit.	✔
ROL *reg/mem8*,CL	D2 /0	Rotate an 8-bit register or memory operand left the number of bits specified in the CL register.	✔
ROL *reg/mem8*,*imm8*	C0 /0 *ib*	Rotate an 8-bit register or memory operand left the number of bits specified by an 8-bit immediate value.	✔
ROL *reg/imm16*,1	D1 /0	Rotate a 16-bit register or memory operand left 1 bit.	✔
ROL *reg/mem16*,CL	D3 /0	Rotate a 16-bit register or memory operand left the number of bits specified in the CL register.	✔
ROL *reg/mem16*,*imm8*	C1 /0 *ib*	Rotate a 16-bit register or memory operand left the number of bits specified by an 8-bit immediate value.	✔
ROL *reg/imm32*,1	D1 /0	Rotate a 32-bit register or memory operand left 1 bit.	✔
ROL *reg/mem32*,CL	D3 /0	Rotate a 32-bit register or memory operand left the number of bits specified in the CL register.	✔
ROL *reg/mem32*,*imm8*	C1 /0 *ib*	Rotate a 32-bit register or memory operand left the number of bits specified by an 8-bit immediate value.	✔
ROL *reg/imm64*,1	D1 /0	Rotate a 64-bit register or memory operand left 1 bit.	✔
ROL *reg/mem64*,CL	D3 /0	Rotate a 64-bit register or memory operand left the number of bits specified in the CL register.	✔
ROL *reg/mem64*,*imm8*	C1 /0 *ib*	Rotate a 64-bit register or memory operand left the number of bits specified by an 8-bit immediate value.	✔
ROR *reg/imm8*,1	D0 /0	Rotate an 8-bit register or memory operand right 1 bit.	✔
ROR *reg/mem8*,CL	D2 /0	Rotate an 8-bit register or memory operand right the number of bits specified in the CL register.	✔
ROR *reg/mem8*,*imm8*	C0 /0 *ib*	Rotate an 8-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
ROR *reg/imm16*,1	D1 /0	Rotate a 16-bit register or memory operand left 1 bit.	✔
ROR *reg/mem16*,CL	D3 /0	Rotate a 16-bit register or memory operand right the number of bits specified in the CL register.	✔
ROR *reg/mem16*,*imm8*	C1 /0 *ib*	Rotate a 16-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
ROR *reg/imm32*,1	D1 /0	Rotate a 32-bit register or memory operand left 1 bit.	✔
ROR *reg/mem32*,CL	D3 /0	Rotate a 32-bit register or memory operand right the number of bits specified in the CL register.	✔
ROR *reg/mem32,imm8*	C1 /0 *ib*	Rotate a 32-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
ROR *reg/imm64*,1	D1 /0	Rotate a 64-bit register or memory operand right 1 bit.	✔
ROR *reg/mem64*,CL	D3 /0	Rotate a 64-bit register or memory operand right the number of bits specified in the CL register.	✔
ROR *reg/mem64,imm8*	C1 /0 *ib*	Rotate a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
SAHF	9E	Loads the sign flag, the zero flag, the auxiliary flag, the parity flag, and the carry flag from the AH register into the lower 8 bits of the EFLAGS register.	✔
SAL *reg/mem8*,1	D0 /4	Shift an 8-bit register or memory location left 1 bit.	✔
SAL *reg/mem8*,CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.	✔
SAL *reg/mem8,imm8*	C0 /4 *ib*	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SAL *reg/mem16*,1	D1 /4	Shift a 16-bit register or memory location left 1 bit.	✔
SAL *reg/mem16*,CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.	✔
SAL *reg/mem16,imm8*	C1 /4 *ib*	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SAL *reg/mem32*,1	D1 /4	Shift a 32-bit register or memory location left 1 bit.	✔
SAL *reg/mem32*,CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.	✔
SAL *reg/mem32,imm8*	C1 /4 *ib*	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SAL *reg/mem64*,1	D1 /4	Shift a 64-bit register or memory location left 1 bit.	✔
SAL *reg/mem64*,CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.	✔
SAL *reg/mem64,imm8*	C1 /4 *ib*	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem8*,1	D0 /4	Shift an 8-bit register or memory location left 1 bit.	✔
SHL *reg/mem8*,CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.	✔
SHL *reg/mem8,imm8*	C0 /4 *ib*	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem16*,1	D1 /4	Shift a 16-bit register or memory location left 1 bit.	✔
SHL *reg/mem16*,CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SHL *reg/mem16*,*imm8*	C1 /4 *ib*	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem32*,1	D1 /4	Shift a 32-bit register or memory location left 1 bit.	✔
SHL *reg/mem32*,CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.	✔
SHL *reg/mem32*,*imm8*	C1 /4 *ib*	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem64*,1	D1 /4	Shift a 64-bit register or memory location left 1 bit.	✔
SHL *reg/mem64*,CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.	✔
SHL *reg/mem64*,*imm8*	C1 /4 *ib*	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SAR *reg/mem8*,1	D0 /7	Shift a signed 8-bit register or memory operand right 1 bit.	✔
SAR *reg/mem8*,CL	D2 /7	Shift a signed 8-bit register or memory operand right the number of bits specified in the CL register.	✔
SAR *reg/mem8*,*imm8*	C0 /7 *ib*	Shift a signed 8-bit register or memory location right the number of bits specified by an 8-bit immediate value.	✔
SAR *reg/mem16*,1	D1 /7	Shift a signed 16-bit register or memory operand right 1 bit.	✔
SAR *reg/mem16*,CL	D3 /7	Shift a signed 16-bit register or memory operand right the number of bits specified in the CL register.	✔
SAR *reg/mem16*,*imm8*	C1 /7 *ib*	Shift a signed 16-bit register or memory location right the number of bits specified by an 8-bit immediate value.	✔
SAR *reg/mem32*,1	D1 /7	Shift a signed 32-bit register or memory location right 1 bit.	✔
SAR *reg/mem32*,CL	D3 /7	Shift a signed 32-bit register or memory operand right the number of bits specified in the CL register.	✔
SAR *reg/mem32*,*imm8*	C1 /7 *ib*	Shift a signed 32-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
SAR *reg/mem64*,1	D1 /7	Shift a signed 64-bit register or memory operand left 1 bit.	✔
SAR *reg/mem64*,CL	D3 /7	Shift a signed 64-bit register or memory operand right the number of bits specified in the CL register.	✔
SAR *reg/mem64*,*imm8*	C1 /7 *ib*	Shift a signed 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
SBB AL,*imm8*	1C *ib*	Subtract an immediate 8-bit value from the AL register with borrow.	✔
SBB AX,*imm16*	1D *iw*	Subtract an immediate 16-bit value from the AX register with borrow.	✔
SBB EAX,*imm32*	1D *id*	Subtract an immediate 32-bit value from the EAX register with borrow.	✔
SBB RAX,*imm32*	1D *id*	Subtract an immediate 32-bit value from the RAX register with borrow.	✔
SBB *reg/mem8*,*imm8*	80 /3 *ib*	Subtract an immediate 8-bit value from an 8-bit register or memory location with borrow.	✔
SBB *reg/mem16*,*imm16*	80 /3 *iw*	Subtract an immediate 16-bit value from a 16-bit register or memory location with borrow.	✔
SBB *reg/mem32*,*imm32*	81 /3 *id*	Subtract an immediate 32-bit value from a 32-bit register or memory location with borrow.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SBB *reg/mem64,imm32*	81 /3 *id*	Subtract a sign-extended immediate 32-bit value from a 64-bit register or memory location with borrow.	✔
SBB *reg/mem16,imm8*	83 /3 *ib*	Subtract a sign-extended 8-bit immediate value from a 16-bit register or memory location with borrow.	✔
SBB *reg/mem32,imm8*	83 /3 *ib*	Subtract a sign-extended 8-bit immediate value from a 32-bit register or memory location with borrow.	✔
SBB *reg/mem64,imm8*	83 /3 *ib*	Subtract a sign-extended 8-bit immediate value from a 64-bit register or memory location with borrow.	✔
SBB *reg/mem8,reg8*	18 /r	Subtract the contents of an 8-bit register from an 8-bit register or memory location with borrow.	✔
SBB *reg/mem16,reg16*	19 /r	Subtract the contents of a 16-bit register from a 16-bit register or memory location with borrow.	✔
SBB *reg/mem32,reg32*	19 /r	Subtract the contents of a 32-bit register from a 32-bit register or memory location with borrow.	✔
SBB *reg/mem64,reg64*	19 /r	Subtract the contents of a 64-bit register from a 64-bit register or memory location with borrow.	✔
SBB *reg8,reg/mem8*	1A /r	Subtract the contents of an 8-bit register or memory location from the contents of an 8-bit register with borrow.	✔
SBB *reg16,reg/mem16*	1B /r	Subtract the contents of a 16-bit register or memory location from the contents of a 16-bit register with borrow.	✔
SBB *reg32,reg/mem32*	1B /r	Subtract the contents of a 32-bit register or memory location from the contents of a 32-bit register with borrow.	✔
SBB *reg64,reg/mem64*	1B /r	Subtract the contents of a 64-bit register or memory location from the contents of a 64-bit register with borrow.	✔
SCAS *mem8*	AE	Compare the contents of the AL register with the byte at ES:rDI, and then increment or decrement rDI.	✔
SCAS *mem16*	AF	Compare the contents of the AX register with the word at ES:rDI, and then increment or decrement rDI.	✔
SCAS *mem32*	AF	Compare the contents of the EAX register with the doubleword at ES:rDI, and then increment or decrement rDI.	✔
SCAS *mem64*	AF	Compare the contents of the RAX register with the quadword at ES:rDI, and then increment or decrement rDI.	✔
SCASB	AE	Compare the contents of the AL register with the byte at ES:rDI, and then increment or decrement rDI.	✔
SCASW	AF	Compare the contents of the AX register with the word at ES:rDI, and then increment or decrement rDI.	✔
SCASD	AF	Compare the contents of the EAX register with the doubleword at ES:rDI, and then increment or decrement rDI.	✔
SCASQ	AF	Compare the contents of the RAX register with the quadword at ES:rDI, and then increment or decrement rDI.	✔
SETO *reg/mem8*	0F 90	Set byte if overflow (OF = 1).	✔
SETNO *reg/mem8*	0F 91	Set byte if not overflow (OF = 0).	✔
SETB *reg/mem8*	0F 92	Set byte if below (CF = 1).	✔
SETC *reg/mem8*	0F 92	Set byte if carry (CF = 1).	✔
SETNAE *reg/mem8*	0F 92	Set byte if not above or equal (CF = 1).	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SETNB *reg/mem8*	0F 93	Set byte if not below (CF = 0).	✔
SETNC *reg/mem8*	0F 93	Set byte if not carry (CF = 0).	✔
SETAE *reg/mem8*	0F 93	Set byte if above or equal (CF = 0).	✔
SETZ *reg/mem8*	0F 94	Set byte if zero (ZF = 1).	✔
SETE *reg/mem8*	0F 94	Set byte if equal (ZF = 1).	✔
SETNZ *reg/mem8*	0F 95	Set byte if not zero (ZF = 0).	✔
SETNE *reg/mem8*	0F 95	Set byte if not equal (ZF = 0).	✔
SETBE *reg/mem8*	0F 96	Set byte if below or equal (CF = 1 or ZF = 1).	✔
SETNA *reg/mem8*	0F 96	Set byte if not above (CF = 1 or ZF = 1).	✔
SETNBE *reg/mem8*	0F 97	Set byte if not below or equal (CF = 0 and ZF = 0).	✔
SETA *reg/mem8*	0F 97	Set byte if above (CF = 0 and ZF = 0).	✔
SETS *reg/mem8*	0F 98	Set byte if sign (SF = 1).	✔
SETNS *reg/mem8*	0F 99	Set byte if not sign (SF = 0).	✔
SETP *reg/mem8*	0F 9A	Set byte if parity (PF = 1).	✔
SETPE *reg/mem8*	0F 9A	Set byte if parity even (PF = 1).	✔
SETNP *reg/mem8*	0F 9B	Set byte if not parity (PF = 0).	✔
SETPO *reg/mem8*	0F 9B	Set byte if parity odd (PF = 0).	✔
SETL *reg/mem8*	0F 9C	Set byte if less (SF <> OF).	✔
SETNGE *reg/mem8*	0F 9C	Set byte if not greater or equal (SF <> OF).	✔
SETNL *reg/mem8*	0F 9D	Set byte if not less (SF =OF).	✔
SETGE *reg/mem8*	0F 9D	Set byte if greater or equal (SF = OF).	✔
SETLE *reg/mem8*	0F 9E	Set byte if less or equal (ZF = 1 or SF <> OF).	✔
SETNG *reg/mem8*	0F 9E	Set byte if not greater (ZF = 1 or SF <> OF).	✔
SETNLE *reg/mem8*	0F 9F	Set byte if not less or equal (ZF = 0 and SF = OF).	✔
SETG *reg/mem8*	0F 9F	Set byte if greater (ZF = 0 and SF = OF).	✔
SFENCE	0F AE F8	Force strong ordering of (serialized) store operations.	✔
SHL *reg/mem8*,1	D0 /4	Shift an 8-bit register or memory location left 1 bit.	✔
SHL *reg/mem8*,CL	D2 /4	Shift an 8-bit register or memory location left the number of bits specified in the CL register.	✔
SHL *reg/mem8*,*imm8*	C0 /4 *ib*	Shift an 8-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem16*,1	D1 /4	Shift a 16-bit register or memory location left 1 bit.	✔
SHL *reg/mem16*,CL	D3 /4	Shift a 16-bit register or memory location left the number of bits specified in the CL register.	✔
SHL *reg/mem16*,*imm8*	C1 /4 *ib*	Shift a 16-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem32*,1	D1 /4	Shift a 32-bit register or memory location left 1 bit.	✔
SHL *reg/mem32*,CL	D3 /4	Shift a 32-bit register or memory location left the number of bits specified in the CL register.	✔
SHL *reg/mem32*,*imm8*	C1 /4 *ib*	Shift a 32-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHL *reg/mem64*,1	D1 /4	Shift a 64-bit register or memory location left 1 bit.	✔
SHL *reg/mem64*,CL	D3 /4	Shift a 64-bit register or memory location left the number of bits specified in the CL register.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SHL *reg/mem64*,*imm8*	C1 /4 *ib*	Shift a 64-bit register or memory location left the number of bits specified by an 8-bit immediate value.	✔
SHLD *reg/mem16*,*reg16*,*imm8*	0F A4 /r *ib*	Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.	✔
SHLD *reg/mem16*,*reg16*,CL	0F A5 /r	Shift bits of a 16-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.	✔
SHLD *reg/mem32*,*reg32*,*imm8*	0F A4 /r *ib*	Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.	✔
SHLD *reg/me326*,*reg32*,CL	0F A5 /r	Shift bits of a 32-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.	✔
SHLD *reg/mem64*,*reg64*,*imm8*	0F A4 /r *ib*	Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.	✔
SHLD *reg/mem16*,*reg16*,CL	0F A5 /r	Shift bits of a 64-bit destination register or memory operand to the left the number of bits specified in the CL register, while shifting in bits from the second operand.	✔
SHR *reg/mem8*,1	D0 /5	Shift an 8-bit register or memory operand right 1 bit.	✔
SHR *reg/mem8*,CL	D2 /5	Shift an 8-bit register or memory operand right the number of bits specified in the CL register.	✔
SHR *reg/mem8*,*imm8*	C0 /5 *ib*	Shift an 8-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
SHR *reg/mem16*,1	D1 /5	Shift a 16-bit register or memory operand right 1 bit.	✔
SHR *reg/mem16*,CL	D3 /5	Shift a 16-bit register or memory operand right the number of bits specified in the CL register.	✔
SHR *reg/mem16*,*imm8*	C1 /5 *ib*	Shift a 16-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
SHR *reg/mem32*,1	D1 /5	Shift a 32-bit register or memory operand right 1 bit.	✔
SHR *reg/mem32*,CL	D3 /5	Shift a 32-bit register or memory operand right the number of bits specified in the CL register.	✔
SHR *reg/mem32*,*imm8*	C1 /5 *ib*	Shift a 32-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔
SHR *reg/mem64*,1	D1 /5	Shift a 64-bit register or memory operand left 1 bit.	✔
SHR *reg/mem64*,CL	D3 /5	Shift a 64-bit register or memory operand right the number of bits specified in the CL register.	✔
SHR *reg/mem64*,*imm8*	C1 /5 *ib*	Shift a 64-bit register or memory operand right the number of bits specified by an 8-bit immediate value.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SHRD *reg/mem16,reg16,imm8*	0F AC /r *ib*	Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.	✔
SHRD *reg/mem16,reg16*,CL	0F AD /r	Shift bits of a 16-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.	✔
SHRD *reg/mem32,reg32,imm8*	0F AC /r *ib*	Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.	✔
SHRD *reg/me326,reg32*,CL	0F AD /r	Shift bits of a 32-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.	✔
SHRD *reg/mem64,reg64,imm8*	0F AC /r *ib*	Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in an 8-bit immediate value, while shifting in bits from the second operand.	✔
SHRD *reg/mem16,reg16*,CL	0F AD /r	Shift bits of a 64-bit destination register or memory operand to the right the number of bits specified in the CL register, while shifting in bits from the second operand.	✔
STC	F9	Set the carry flag (CF) to 1.	✔
STD	FD	Set the direction flag (DF) to 1.	✔
STOS *reg8*	AA	Store the contents of the AL register to ES:rDI, and then increment or decrement rDI.	✔
STOS *reg16*	AB	Store the contents of the AX register to ES:rDI, and then increment or decrement rDI.	✔
STOS *reg32*	AB	Store the contents of the EAX register to ES:rDI, and then increment or decrement rDI.	✔
STOS *reg64*	AB	Store the contents of the RAX register to ES:rDI, and then increment or decrement rDI.	✔
STOSB	AA	Store the contents of the AL register to ES:rDI, and then increment or decrement rDI.	✔
STOSW	AB	Store the contents of the AX register to ES:rDI, and then increment or decrement rDI.	✔
STOSD	AB	Store the contents of the EAX register to ES:rDI, and then increment or decrement rDI.	✔
STOSQ	AB	Store the contents of the RAX register to ES:rDI, and then increment or decrement rDI.	✔
SUB AL,*imm8*	2C *ib*	Subtract an immediate 8-bit value from the AL register and store the result in AL.	✔
SUB AX,*imm16*	2D *iw*	Subtract an immediate 16-bit value from the AX register and store the result in AX.	✔
SUB EAX,*imm32*	2D *id*	Subtract an immediate 32-bit value from the EAX register and store the result in EAX.	✔
SUB RAX,*imm32*	2D *id*	Subtract a sign-extended immediate 32-bit value from the RAX register and store the result in RAX.	✔
SUB *reg/mem8,imm8*	80 /5 *ib*	Subtract an immediate 8-bit value from an 8-bit destination register or memory location.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SUB *reg/mem16,imm16*	81 /5 *iw*	Subtract an immediate 16-bit value from a 16-bit destination register or memory location.	✔
SUB *reg/mem32,imm32*	81 /5 *id*	Subtract an immediate 32-bit value from a 32-bit destination register or memory location.	✔
SUB *reg/mem64,imm32*	81 /5 *id*	Subtract a sign-extended immediate 32-bit value from a 64-bit destination register or memory location.	✔
SUB *reg/mem16,imm8*	83 /5 *ib*	Subtract a sign-extended immediate 8-bit value from a 16-bit register or memory location.	✔
SUB *reg/mem32,imm8*	83 /5 *ib*	Subtract a sign-extended immediate 8-bit value from a 32-bit register or memory location.	✔
SUB *reg/mem64,imm8*	83 /5 *ib*	Subtract a sign-extended immediate 8-bit value from a 64-bit register or memory location.	✔
SUB *reg/mem8,reg8*	28 /r	Subtract the contents of an 8-bit register from an 8-bit destination register or memory location.	✔
SUB *reg/mem16,reg16*	29 /r	Subtract the contents of a 16-bit register from a 16-bit destination register or memory location.	✔
SUB *reg/mem32,reg32*	29 /r	Subtract the contents of a 32-bit register from a 32-bit destination register or memory location.	✔
SUB *reg/mem64,reg64*	29 /r	Subtract the contents of a 64-bit register from a 64-bit destination register or memory location.	✔
SUB *reg8,reg/mem8*	2A /r	Subtract the contents of an 8-bit register or memory operand from an 8-bit destination register.	✔
SUB *reg16,reg/mem16*	2B /r	Subtract the contents of a 16-bit register or memory operand from a 16-bit destination register.	✔
SUB *reg32,reg/mem32*	2B /r	Subtract the contents of a 32-bit register or memory operand from a 32-bit destination register.	✔
SUB *reg64,reg/mem64*	2B /r	Subtract the contents of a 64-bit register or memory operand from a 64-bit destination register.	✔
TEST AL,*imm8*	AB *ib*	AND an immediate 8-bit value with the contents of the AL register and set rFLAGS to reflect the result.	✔
TEST AX,*imm16*	A9 *iw*	AND an immediate 16-bit value with the contents of the AX register and set rFLAGS to reflect the result.	✔
TEST EAX,*imm32*	A9 *id*	AND an immediate 32-bit value with the contents of the EAX register and set rFLAGS to reflect the result.	✔
TEST RAX,*imm32*	A9 *id*	AND a sign-extened immediate 32-bit value with the contents of the RAX register and set rFLAGS to reflect the result.	✔
TEST *reg/mem8,imm8*	F6 /0 *ib*	AND an immediate 8-bit value with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result.	✔
TEST *reg/mem16,imm16*	F7 /0 *iw*	AND an immediate 16-bit value with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result.	✔
TEST *reg/mem32,imm32*	F7 /0 *id*	AND an immediate 32-bit value with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result.	✔
TEST *reg/mem64,imm32*	F7 /0 *id*	AND a sign-extened immediate 32-bit value with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result.	✔
TEST *reg/mem8,reg8*	84 /r	AND the contents of an 8-bit register with the contents of an 8-bit register or memory operand and set rFLAGS to reflect the result.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
TEST *reg/mem16*,*reg16*	85 /r	AND the contents of a 16-bit register with the contents of a 16-bit register or memory operand and set rFLAGS to reflect the result.	✔
TEST *reg/mem32*,*reg32*	85 /r	AND the contents of a 32-bit register with the contents of a 32-bit register or memory operand and set rFLAGS to reflect the result.	✔
TEST *reg/mem64*,*reg64*	85 /r	AND the contents of a 64-bit register with the contents of a 64-bit register or memory operand and set rFLAGS to reflect the result.	✔
XADD *reg/mem8*,*reg8*	0F C0 /r	Exchange the contents of an 8-bit register with the contents of 8-bit destination register or memory operand and load their sum into the destination.	✔
XADD *reg/mem16*,*reg16*	0F C1 /r	Exchange the contents of a 16-bit register with the contents of 16-bit destination register or memory operand and load their sum into the destination.	✔
XADD *reg/mem32*,*reg32*	0F C1 /r	Exchange the contents of a 32-bit register with the contents of 32-bit destination register or memory operand and load their sum into the destination.	✔
XADD *reg/mem64*,*reg64*	0F C1 /r	Exchange the contents of a 64-bit register with the contents of 64-bit destination register or memory operand and load their sum into the destination.	✔
XCHG AX,*reg16*	90 *+rw*	Exchange the contents of AX register with the contents of a 16-bit register.	✔
XCHG *reg16*,AX	90 *+rw*	Exchange the contents of a 16-bit register with the contents of the AX register.	✔
XCHG AX,*reg32*	90 *+rd*	Exchange the contents of EAX register with the contents of a 32-bit register.	✔
XCHG *reg32*,AX	90 *+rd*	Exchange the contents of a 32-bit register with the contents of the EAX register.	✔
XCHG RAX,*reg64*	90 *+rq*	Exchange the contents of RAX register with the contents of a 64-bit register.	✔
XCHG *reg64*,RAX	90 *+rq*	Exchange the contents of a 64-bit register with the contents of the RAX register.	✔
XCHG *reg/mem8*,*reg8*	86 /r	Exchange the contents of an 8-bit register with the contents of an 8-bit register or memory operand.	✔
XCHG *reg8*,*reg/mem8*	86 /r	Exchange the contents of an 8-bit register or memory operand with the contents of an 8-bit register.	✔
XCHG *reg/mem16*,*reg16*	87 /r	Exchange the contents of a 16-bit register with the contents of a 16-bit register or memory operand.	✔
XCHG *reg16*,*reg/mem16*	87 /r	Exchange the contents of a 16-bit register or memory operand with the contents of a 16-bit register.	✔
XCHG *reg/mem32*,*reg32*	87 /r	Exchange the contents of a 32-bit register with the contents of a 32-bit register or memory operand.	✔
XCHG *reg32*,*reg/mem32*	87 /r	Exchange the contents of a 32-bit register or memory operand with the contents of a 32-bit register.	✔
XCHG *reg/mem64*,*reg64*	87 /r	Exchange the contents of a 64-bit register with the contents of a 64-bit register or memory operand.	✔
XCHG *reg64*,*reg/mem64*	87 /r	Exchange the contents of a 64-bit register or memory operand with the contents of a 64-bit register.	✔
XLAT *mem8*	D7	Set AL to the contents of DS:[rBX + unsigned AL].	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
XLATB	D7	Set AL to the contents of DS:[rBX + unsigned AL].	✔
XOR AL,*imm8*	34 *ib*	XOR the contents of AL with an immediate 8-bit operand and store the result in AL.	✔
XOR AX,*imm16*	35 *iw*	XOR the contents of AX with an immediate 16-bit operand and store the result in AX.	✔
XOR EAX,*imm32*	35 *id*	XOR the contents of EAX with an immediate 32-bit operand and store the result in EAX.	✔
XOR RAX,*imm32*	35 *id*	XOR the contents of RAX with a sign-extended immediate 32-bit operand and store the result in AX.	✔
XOR *reg/mem8*,*imm8*	80 /6 *ib*	XOR the contents of an 8-bit destination register or memory operand with an 8-bit immediate value and store the result in the destination.	✔
XOR *reg/mem16*,*imm16*	81 /6 *iw*	XOR the contents of a 16-bit destination register or memory operand with a 16-bit immediate value and store the result in the destination.	✔
XOR *reg/mem32*,*imm32*	81 /6 *id*	XOR the contents of a 32-bit destination register or memory operand with a 32-bit immediate value and store the result in the destination.	✔
XOR *reg/mem64*,*imm32*	81 /6 *id*	XOR the contents of a 64-bit destination register or memory operand with a sign-extended 32-bit immediate value and store the result in the destination.	✔
XOR *reg/mem16*,*imm8*	83 /6 *ib*	XOR the contents of a 16-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.	✔
XOR *reg/mem32*,*imm8*	83 /6 *ib*	XOR the contents of a 32-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.	✔
XOR *reg/mem64*,*imm8*	83 /6 *ib*	XOR the contents of a 64-bit destination register or memory operand with a sign-extended 8-bit immediate value and store the result in the destination.	✔
XOR *reg/mem8*,*reg8*	30 /r	XOR the contents of an 8-bit destination register or memory operand with the contents of an 8-bit register and store the result in the destination.	✔
XOR *reg/mem16*,*reg16*	31 /r	XOR the contents of a 16-bit destination register or memory operand with the contents of a 16-bit register and store the result in the destination.	✔
XOR *reg/mem32*,*reg32*	31 /r	XOR the contents of a 32-bit destination register or memory operand with the contents of a 32-bit register and store the result in the destination.	✔
XOR *reg/mem64*,*reg64*	31 /r	XOR the contents of a 64-bit destination register or memory operand with the contents of a 64-bit register and store the result in the destination.	✔
XOR *reg8*,*reg/mem8*	32 /r	XOR the contents of an 8-bit destination register with the contents of an 8-bit register or memory operand and store the result in the destination.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
XOR *reg16,reg/mem16*	33 /r	XOR the contents of a 16-bit destination register with the contents of a 16-bit register or memory operand and store the result in the destination.	✔
XOR *reg32,reg/mem32*	33 /r	XOR the contents of a 32-bit destination register with the contents of a 32-bit register or memory operand and store the result in the destination.	✔
XOR *reg64,reg/mem64*	33 /r	XOR the contents of a 64-bit destination register with the contents of a 64-bit register or memory operand and store the result in the destination.	✔

Table 15-8: General-Purpose Instruction Reference

A.6.3 System Instructions

This chapter describes the function, mnemonic syntax and opcodes that the simulator simulates. The system instructions are used to establish the operating mode, access processor resources, handle program and system errors, and manage memory. Many of these instructions can only be executed by privileged software, such as the operating system kernel and interrupt handlers, that run at the highest privilege level. Only system instructions can access certain processor resources, such as the control registers, model-specific register, and debug registers.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
ARPL *reg/mem16,reg16*	63 /r	Adjust the RPL of a destination segment selector to a level not less than the RPL of the segment selector specifies in the 16-bit source register.	✔[1]
CLI	FA	Clear the interrupt flag (IF) to zero.	✔
CLTS	0F 06	Clear the task-switched (TS) flag in CR0 to 0.	✔
HLT	F4	Halt instruction execution.	✔
INT 3	CC	Trap to debugger at interrupt 3.	⚠[2]
INVD	0F 08	Flush internal caches and trigger external cache flushes.	✔
INVLPG *mem8*	0F 01 /7	Invalidate the TLB entry for the page containing a specified memory location.	✔
IRET	CF	Return from interrupt (16-bit operand size).	⚠[3]
IRETD	CF	Return from interrupt (32-bit operand size).	⚠[3]
IRETQ	CF	Return from interrupt (64-bit operand size).	⚠[3]
LAR *reg16,reg/mem16*	0F 02 /r	Reads the GDT/LDT descriptor referenced by the 16-bit source operand masks the attributes with FF00h and saves the result in the 16-bit destination register.	✔
LAR *reg32,reg/mem16*	0F 02 /r	Reads the GDT/LDT descriptor referenced by the 16-bit source operand masks the attributes with 00FFFF00h and saves the result in the 32-bit destination register.	✔

[1] In 64-bit mode, this opcode (0x63) is used for the MOVSXD instruction.
[2] See Section A.6.3.1, "*INT – Interrupt to Vector*", on page 225.
[3] See Section A.6.3.2, "*IRET – Return from Interrupt*", on page 225.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
LAR *reg64,reg/mem16*	0F 02 /r	Reads the GDT/LDT descriptor referenced by the 16-bit source operand, masks the attributes with 00FFFF00h and saves the result in the 64-bit destination register.	✔
LGDT *mem16:32*	0F 01 /2	Loads *mem16:32* into the global descriptor table register.	✔
LGDT *mem16:64*	0F 01 /2	Loads *mem16:64* into the global descriptor table register.	✔
LIDT *mem16:32*	0F 01 /3	Loads *mem16:32* into the interrupt descriptor table register.	✔
LIDT *mem16:64*	0F 01 /3	Loads *mem16:64* into the interrupt descriptor table register.	✔
LLDT *reg/mem16*	0F 00 /2	Load the 16-bit segment selector into the local descriptor table register and load the LDT descriptor from the GDT.	✔
LMSW *reg/mem16*	0F 01 /6	Loads the lower 4 bits of the source into the lower 4 bits of CR0.	✔
LSL *reg16,reg/mem16*	0F 03 /r	Loads a 16-bit general-purpose register with the segment limit or a selector specified in a 16-bit memory or register operand.	✔
LSL *reg32,reg/mem16*	0F 03 /r	Loads a 32-bit general-purpose register with the segment limit or a selector specified in a 16-bit memory or register operand.	✔
LSL *reg64,reg/mem16*	0F 03 /r	Loads a 64-bit general-purpose register with the segment limit or a selector specified in a 16-bit memory or register operand.	✔
LTR *reg/mem16*	0F 00 /3	Load the 16-bit segment selector into the task register and load the TSS descriptor from the GDT.	✔
MOV *CRn,reg32*	0F 22 /r	Move the contents of a 32-bit register to CR*n*.	✔
MOV *CRn,reg64*	0F 22 /r	Move the contents of a 64-bit register to CR*n*.	✔
MOV *reg32,CRn*	0F 20 /r	Move the contents of CR*n* to a 32-bit register.	✔
MOV *reg64,CRn*	0F 20 /r	Move the contents of CR*n* to a 64-bit register.	✔
MOV *DRn,reg32*	0F 21 /r	Move the contents of a 32-bit register to DR*n*.	✔
MOV *DRn,reg64*	0F 21 /r	Move the contents of a 64-bit register to DR*n*.	✔
MOV *reg32,DRn*	0F 23 /r	Move the contents of DR*n* to a 32-bit register.	✔
MOV *reg64,DRn*	0F 23 /r	Move the contents of DR*n* to a 64-bit register.	✔
RDMSR	0F 32	Copy MSR specified by ECX into EDX:EAX.	✔
RDPMC	0F 33	Copy the performance monitor counter specified by ECX into EDX:EAX.	✔
RDTSC	0F 31	Copy the time-stamp counter into EDX:EAX.	✔
RSM	0F AA	Resume operation of an interrupted program.	✔
SGDT *mem16:32*	0F 01 /0	Store global descriptor table register to memory.	✔
SGDT *mem16:64*	0F 01 /0	Store global descriptor table register to memory.	✔
SIDT *mem16:32*	0F 01 /1	Store interrupt descriptor table register to memory.	✔
SIDT *mem16:64*	0F 01 /1	Store interrupt descriptor table register to memory.	✔
SLDT *reg16*	0F 00 /0	Store the segment selector from the local descriptor table register to a 16-bit register.	✔
SLDT *reg32*	0F 00 /0	Store the segment selector from the local descriptor table register to a 32-bit register.	✔
SLDT *reg64*	0F 00 /0	Store the segment selector from the local descriptor table register to a 64-bit register.	✔
SLDT *mem16*	0F 00 /0	Store the segment selector from the local descriptor table register to a 16-bit memory location.	✔
SMSW *reg16*	0F 01 /4	Store the low 16 bits of CR0 to a 16-bit register.	✔
SMSW *reg32*	0F 01 /4	Store the low 32 bits of CR0 to a 32-bit register.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
SMSW *reg64*	0F 01 /4	Store the entire 64 bits of CR0 to a 64-bit register.	✔
SMSW *mem16*	0F 01 /4	Store the low 16 bits of CR0 to memory.	✔
STI	FB	Set interrupt flag (IF) to 1.	✔
STR *reg16*	0F 00 /1	Store the segment selector from the task register to a 16-bit general-purpose register.	✔
STR *reg32*	0F 00 /1	Store the segment selector from the task register to a 32-bit general-purpose register.	✔
STR *reg64*	0F 00 /1	Store the segment selector from the task register to a 64-bit general-purpose register.	✔
STR *mem16*	0F 00 /1	Store the segment selector from the task register to a 16-bit memory location.	✔
SWAPGS	0F 01 F8	Exchange GS base with KernelGSBase MSR.	✔
SYSCALL	0F 05	Call operating system.	✔
SYSENTER	0F 34	Call operating system.	✔
SYSEXIT	0F 35	Return from operating system.	✔
SYSRET	0F 07	Return from operating system.	✔
UD2	0F 08	Raise an invalid opcode exception.	✔
VERR *reg/mem16*	0F 00 /4	Set the zero flag (ZF) to 1 if the segment selected can be read.	✔
VERW	0F 00 /5	Set the zero flag (ZF) to 1 if the segment selected can be written.	✔
WBINVD	0F 09	Write modified cache lines to main memory, invalidate internal caches, and trigger external cache flushes.	✔
WRMSR	0F 30	Write EDX:EAX to the MSR specified by ECX.	✔

Table 15-9: System Instruction Reference

A.6.3.1 INT – Interrupt to Vector

Opcode	Instruction	Description
CD	INT *imm8*	Interrupt to Vector.
CC	INT 3	Interrupt to Debug Vector.

- Interrupt to task-gate is not implemented. An attempt to execute an interrupt to task-gate results in a '*FeatureNotImplemented'* exception and the simulation will be stopped.
- When delivering an exception in an attempt to deliver a hardware interrupt the simulation will not push the resume-flag (RF) onto the stack.
- Always clears VM, NT, TF, and RF bits in rFLAGS.

A.6.3.2 IRET – Return from Interrupt

Opcode	Instruction	Description
CF	IRET, IRETD, IRETQ	Return from interrupt

The simulator does not support nested task-switching using the rFLAGS *nested-task bit* (NT) and the TSS back-link field. An interrupt return (IRET) to the previous task (nested-task) will result in a '*FeatureNotImplemented'* exception and the simulation will be stopped.

A.6.4 Virtualization Instruction Reference

For more information on Virtualization Technology, see AMD Publication #33047, *AMD64 Virtualization Technology*.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
CLGI	0F 01 DD	Clear Global Interrupt Flag.	✔
INVLPGA	0F 01 DF	Invalidates the TLB mapping for the virtual page specified in rAX and the ASID specified in ECX.	✔
MOV *reg32,CR8*	F0 20 /r	Alternate notation for move from CR8 to register.	✔
MOV *reg64,CR8*	F0 20 /r	Alternate notation for move register to CR8.	✔
MOV *CR8,reg32*	F0 22 /r	Alternate notation for move from CR8 to register.	✔
MOV *CR8,reg64*	F0 22 /r	Alternate notation for move register to CR8.	✔
SKINIT	0F 01 DE	Secure initialization and jump, with attestation.	✔
STGI	0F 01 DC	Set Global Interrupt Flag.	✔
VMLOAD	0F 01 DA	Load State from VMCB.	✔
VMCALL	0F 01 D9	Call VMM.	✔
VMRUN	0F 01 D8	Run Virtual Machine.	✔
VMSAVE	0F 01 DB	Save State to VMCB.	✔

A.6.5 64-Bit Media Instruction Reference

These instructions described in this section operate on data located in the 64-bit MMX registers. Most of the instructions operate in parallel on sets of packed elements called vectors, although some operate on scalars. The instructions define both integer and floating-point operations, and include the legacy MMX instructions and the AMD extensions to the MMX instruction set.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
CVTPD2PI *mmx,xmm2/m128*	66 0F 2D /r	Converts packed double-precision floating-point values in an XMM register or 128-bit memory location to packed doubleword integers values in the destination MMX™ register.	✔
CVTPI2PD x*mm,mmx/m64*	66 0F 2A /r	Converts two packed doubleword integer values in a MMX™ register or 64-bit memory location to two packed double-precision floating-point values in the destination XMM register.	✔
CVTPI2PS *mmx,xmm2/m128*	0F 2A /r	Converts packed doubleword integer values in a MMX™ register or 64-bit memory location to single-precision floating-point values in the destination XMM register.	✔

A.6.6 3DNow!™ Instruction Set

This chapter describes the 3DNow! Instruction Set that the simulator supports and simulates. 3DNow! Technology is a group of new instructions that opens the traditional processing bottlenecks for floating-point-intensive and multimedia applications.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
FEMMS	0F 0E	Fast Enter/Exit of the MMX or floating-point state.	✔

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
PAVGUSB *mmreg1,mmreg2/m64*	0F 0F /BF	Average of unsigned packed 8-bit values.	✔
PF2ID *mmreg1,mmreg2/m64*	0F 0F /1D	Converts packed floating-point operand or packed 32-bit integer.	✔
PFACC *mmreg1,mmreg2/m64*	0F 0F /AE	Floating-point accumulate.	✔
PFADD *mmreg1,mmreg2/m64*	0F 0F /9E	Packed, floating-point addition.	✔
PFCMPEQ *mmreg1,mmreg2/m64*	0F 0F /B0	Packed floating-point comparison, equal to.	✔
PFCMPPGE *mmreg1,mmreg2/m64*	0F 0F /90	Packed floating-point comparison, greater than or equal to.	✔
PFCMPGT *mmreg1,mmreg2/m64*	0F 0F /A0	Packed floating-point comparison, greater than.	✔
PFMAX *mmreg1,mmreg2/m64*	0F 0F /A4	Packed floating-point maximum.	✔
PFMIN *mmreg1,mmreg2/m64*	0F 0F /94	Packed floating-point minimum.	✔
PFMUL *mmreg1,mmreg2/m64*	0F 0F /B4	Packed floating-point multiplication.	✔
PFRCP *mmreg1,mmreg2/m64*	0F 0F /96	Packed floating-point approximation.	✔
PFRCPIT1 *mmreg1,mmreg2/m64*	0F 0F /A6	Packed floating-point reciprocal, first iteration step.	✔
PFRCPIT2 *mmreg1,mmreg2/m64*	0F 0F /B6	Packed floating-point reciprocal, second iteration step.	✔
PFRSQIT1 *mmreg1,mmreg2/m64*	0F 0F /A7	Packed floating-point reciprocal, square root, first iteration step.	✔
PFRSQRT *mmreg1,mmreg2/m64*	0F 0F /97	Packed floating-point reciprocal, square root approximation.	✔
PFSUB *mmreg1,mmreg2/m64*	0F 0F /9A	Packed, floating-point subtraction.	✔
PFSUBR *mmreg1,mmreg2/m64*	0F 0F /AA	Packed, floating-point reverse subtraction.	✔
PI2FD *mmreg1,mmreg2/m64*	0F 0F /0D	Packed 32-bit integer to floating-point conversion.	✔
PMULHRW *mmreg1,mmreg2/m64*	0F 0F /B7	Multiply signed packed 16-bit values with rounding and store the high 16 bits.	✔
PREFETCH/PREFETCHW	0F 0D	Prefetch processor cache line into L1 data cache (Dcache).	✔

Table 15-10: 3DNow!™ Instruction Reference

A.6.7 Extension to the 3DNow! Instruction Set

This section describes the five new DSP instructions added to the 3DNow! Instruction set.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
PF2IW *mmreg1,mmreg2/m64*	0F 0F /1C	Packed floating-point to integer word conversion with sign extend.	✔
PFNACC *mmreg1,mmreg2/m64*	0F 0F /8A	Packed floating-point negative accumulate.	✔
PFPNACC *mmreg1,mmreg2/m64*	0F 0F /8E	Packed floating-point mixed positive-negative accumulate.	✔
PI2FW *mmreg1,mmreg2/m64*	0F 0F /0C	Packed 16-bit integer to floating-point conversion.	✔
PSWAPD *mmreg1,mmreg2/m64*	0F 0F /BB	Packed swap double word.	✔

Table 15-11: Extension to 3DNow! Instruction Reference

A.6.8 Prescott New Instructions

Prescott New Instruction technology for the x64 architecture is a set of 13 new instructions that accelerate performance of Streaming SIMD Extension technology, Streaming SIMD Extension 2 technology, and x87-FP math capabilities. The new technology is compatible with existing software and should run correctly, without modification. The thirteen new instructions are summarized in the following section. For detailed information on each instruction refer to a complete Instruction Set Reference.

Instruction			Supported
Mnemonic	**Opcode**	**Description**	
ADDSUBPD *xmm1,xmm2/m128*	66 0F D0 /r	Add/Subtract packed double-precision floating-point number from XMM2/Mem to XMM1.	✔
ADDSUBPS *xmm1,xmm2/m128*	F2 0F D0 /r	Add/Subtract packed single-precision floating-point number from XMM2/Mem to XMM1.	✔
FISTTP *m16int*	DF /1	Store ST as a signed integer (truncate) in m16int and pop ST.	⚠
FISTTP *m32int*	DB /1	Store ST as a signed integer (truncate) in m32int and pop ST.	⚠
FISTTP *m64int*	DD /1	Store ST as a signed integer (truncate) in m16int and pop ST.	⚠
HADDPD *xmm1,xmm2/m128*	66 0F 7C /r	Add horizontally packed double-precision floating-point numbers from XMM2/Mem to XMM1.	✔
HADDPS *xmm1,xmm2/m128*	F2 0F 7C /r	Add horizontally packed single-precision floating-point numbers from XMM2/Mem to XMM1.	✔
HSUBPD *xmm1,xmm2/m128*	66 0F 7D /r	Subtract horizontally packed double-precision floating-point numbers from XMM2/Mem to XMM1.	✔
HSUBPS *xmm1,xmm2/m128*	F2 0F 7D /r	Subtract horizontally packed single-precision floating-point numbers from XMM2/Mem to XMM1.	✔
LDDQU *xmm,m128*	F2 0F F0 /r	Load 128 bits from Memory to XMM register.	✔
MONITOR *EAX,ECX,EDX*	0F 01 C8	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be of a write-back memory caching type.	✘[1]
MOVDDUP *xmm1,xmm2/m64*	F2 0F 12 /r	Move 64 bits representing the lower double-precision data element from XMM2/Mem to XMM1 register and duplicate.	✔
MOVSHDUP *xmm1,xmm2/m128*	F3 0F 16 /r	Move 128 bits representing packed single-precision data elements from XMM2/Mem to XMM1 register and duplicate high.	✔
MOVSLDUP *xmm1,xmm2/m128*	F3 0F 12 /r	Move 128 bits representing packed single-precision data elements from XMM2/Mem to XMM1 register and duplicate low.	✔
MWAIT EAX,ECX	0F 01 C9	A hint that allows the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class events.	✘[2]

Table 15-12: Prescott New Instruction Reference

A.6.8.1 MONITOR – Setup Monitor Address

Opcode	Instruction	Description
0F 01 C8	MONITOR	Setup Monitor Address.

The simulator does not recognize this instruction. Therefore the simulator generates an invalid-opcode exception.

[1] See Section A.6.8.1, "*MONITOR – Setup Monitor Address*", on page 228.
[2] See Section A.6.8.2, "*MWAIT – Monitor Wait*", on page 229.

A.6.8.2 MWAIT – Monitor Wait

Opcode	Instruction	Description
0F 01 C9	MWAIT	Monitor Wait.

The simulator does not recognize this instruction. Therefore the simulator generates an invalid-opcode exception.

A.7 Automation Commands

The simulator can be controlled externally through a scripting interface by issuing automation commands. These commands are directed toward either the shell, or toward any device that is part of the currently loaded BSD. Automation commands are plain ASCII text, and are sent to the simulator's automation interface. The method for sending automation commands to the interface, and for retrieving the response, is host dependent on the host OS.

Figure 15-1 shows the simulators *Console Window*. The *Console Window* is the user interface to the simulators automation interface. All automation commands can be send from the *Console Window* to the simulators automation interface, as explained in the following sections.

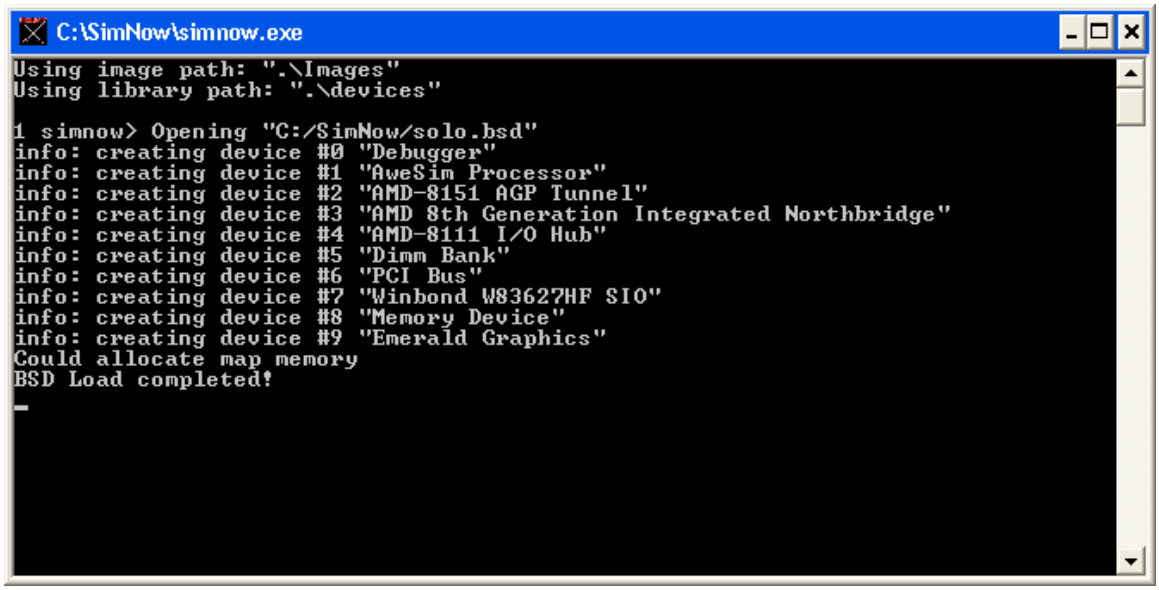

C:\SimNow\simnow.exe
Using image path: ".\Images"
Using library path: ".\devices"
1 simnow> Opening "C:/SimNow/solo.bsd"
info: creating device #0 "Debugger"
info: creating device #1 "AweSim Processor"
info: creating device #2 "AMD-8151 AGP Tunnel"
info: creating device #3 "AMD 8th Generation Integrated Northbridge"
info: creating device #4 "AMD-8111 I/O Hub"
info: creating device #5 "Dimm Bank"
info: creating device #6 "PCI Bus"
info: creating device #7 "Winbond W83627HF SIO"
info: creating device #8 "Memory Device"
info: creating device #9 "Emerald Graphics"
Could allocate map memory
BSD Load completed!

Figure 15-1: Console Window

The automation commands are sent to a specific device by starting the command with the name of the device, followed by a period. For example, to send the *Modules* command to the *shell* device, you would use:

```
1 simnow> shell.modules
```

If more than one device exists in the currently loaded BSD (for example, most BSDs include two IDE controllers), you identify the specific device by following the device name with a colon, and then the number of the device you are interested in. For example, to send the *DVDROMStatus* command to the second IDE controller, you would use:

```
1 simnow> ide:1.DVDROMStatus 0
```

Omitting the colon and the device number causes the simulator to assume device 0. The following two commands are equivalent:

```
1 simnow> ide:0.DVDROMStatus 0
1 simnow> ide.DVDROMStatus 0
```

In addition to the commands supported by the various devices, detailed below, all devices support the *usage* and *ausage* command. These commands return a brief description of each of the commands supported by a specific device. For example, to get a non-alphabetic ordered list of the commands supported by the *shell*, you could send the command:

```
1 simnow> shell.usage
```

To get an alphabetic ordered list of the commands supported by the *shell*, please use the *ausage* command as shown here:

```
1 simnow> shell.ausage
```

To get an overview of all automation commands which are not attached to any specific device enter:

```
1 simnow> help
```

Automation Command	Description
exec <file>	Execute automation commands in [file].
usage	List shell automation commands, same as "*shell.usage*".
newmachine	Create a new SimNow machine, and make that machine the "current" machine for subsequent commands.
switchmachine <n>	Switches the "current" machine to the machine identified by '*n*' the given number.
listmachines	Lists the SimNow machines that currently exist.
exit	Quits the current SimNow machine.
quit	Exits the current SimNow machine.
?	Displays all automation commands which are not attached to any specific device.
help	The same as '?'.

A.7.1 Shell

To list all registered shell commands enter

```
1 simnow> shell.usage
```

Automation Command	Description
ECHO <Value>	Displays *value* to the standard output device (by default, the screen).
Exit	Closes all open GUI components and exits the simulator.

Automation Command	Description
Quit	See *Exit*.
Go	Starts the simulation, see also Section 3.1, "*Tool Bar Buttons*", on page 7.
Stop	Stops the simulation, see also Section 3.1, "*Tool Bar Buttons*", on page 7. The *Stop* command does not return until the simulation has in fact stopped or the stop has failed.
Close	Closes a BSD file that was previously opened.
Open <FileName>	Opens a BSD file.
Modules	Lists all loaded modules.
Running	*Shell.running* returns '*No*' if simulation is currently not running; otherwise it returns '*Yes*'.
Save [<Filename>]	Saves the current system configuration to a file. Default is "*simnow.bsd*"
RunTimeDuration <time>	Runs the simualtion for the given number of microseconds and then stops the simulation.
GetRunTimeDuration	Returns the run time duration in nanoseconds.
ModifyRegistry <key> <value>	*ModifyKey* modifies and updates the given registry key with the given value.
LogConsoleEnabled	*Shell.LogConsoleEnabled* returns '*disabled*' if console logging is disabled; otherwise it returns '*enabled*'.
SetLogConsoleEnabled <0|1>	Enables or disables logging. *Shell.SetLogConsoleEnabled 1* enables logging and *Shell.SetLogConsoleEnabled 0* disables logging.
LogWndEnabled	Returns the Log Window status. The status is '*enabled*' or '*disabled*'.
SetLogWndEnabled <0|1>	Sets the Log Window status to '*enabled*' or '*disabled*'.
LogFile	Returns the current Log file name. Default is '*simnow.log*'.
SetLogFile <filename>	Sets the Log file name.
LogFileEnabled	Returns '*enabled*' if file logging is enabled otherwise it returns '*disabled*'.
SetLogFileEnabled <0|1>	Enables or disabled file logging. 0 disables file logging, 1 enables file logging.
LogDevice <Device Name> <0 | 1>	Enabled (1) or disables (0) device logging for <device>.
LoggingEnabled <Device Name>	Returns the logging status of device <Device Name>. This automation command returns *enabled* or *disabled*.
ErrorLogFile	Returns the current Error Log file name. Default is '*simnow.errlog*'.
SetErrorLogFile <filename>	Sets the Error Log file name.
ErrorLogFileEnabled	Returns '*enabled*' if error file logging is enabled otherwise it returns '*disabled*'.
SetErrorLogFileEnabled <0|1>	Enables or disabled error file logging. 0 disables error file logging; 1 enables error file logging.

Automation Command	Description
Memdump <FileName>	Set the memory dump file name.
Reset	Resets the simulation, see also Section 3.1, "*Tool Bar Buttons*", on page 7.
CreatedDevices	Lists all created devices.
AddDevice <Device Name> [<x> <y>]]	Creates a device and adds the device to the device window at position (x, y). 'x' and 'y' are pixel coordinates inside the device window.
Connections <Device Name>	Lists all connections that a device has.
Connect <Device Name1> [connect point1] [Device Name2] [connect point2]	Connects '*Device Name1'* and '*Device Name2*' using '*connect point1*' and '*connect point2*'.
AvailablePorts <Device Name>	Lists available ports of device '*Device Name*'.
Disconnect <Device Name>	Disconnects all connections of device '*Device Name*'.
DeleteDevice <Device Name>	Deletes device '*Device Name*' from simulated system and removes it from device window.
KnownDevices	Lists all devices that are known by the simulator. These devices are stored in '*devices*\'.
MoveDevice <Device Name> <x> <y>	Moves the specified device 'Device Name' to x/y coordinates in device window. This command only work when GUI mode is active.
New	Creates a new BSD file.
Location	Returns the location/postion (x, y) of the device <Device Name> in the device window. 'x' and 'y' are pixel coordinates inside the device window. For example, *Location "USB JumpDrive"* returns *"USB JumpDrive" 152 382* where *152* is the 'x' coordinate and *382* is the 'y' coordinate.
DumpRegistry	Displays all information stored in SimNow's registry.
SetMPQuantum <time (nanoseconds)>	Sets the time in nanoseconds for a CPU before switching to next CPU in a MP system. Modifying the MP Quantum might have a huge impact on the simulated MP system.
GetMPQuantum	Returns the current MP Quantum value (see also *SetMPQuantum*).
GDB -d [[udp|tcp][::] [<port>]]	Sets up the simulators gdb interface. The default protocol is tcp and the default port is *2222*. If you don't define any parameters the default protocol and port will be used. You can override *tcp* with *udp*. The following example shows how to override the default protocol and port parameters: "*shell.gdb udp::2233"*. The host parameter [::] can't be changed it is always set to localhost. For more information please refer to Section 11.2, "*GDB Interface*", on page 156.

Automation Command	Description
Swap {X86Sim Processor | AweSim Processor}	Switches CPU model from *X86Sim* to *AweSim* or the other way around.
HasModule <module>	Returns '*true*' if module is present; otherwise it returns '*false*'.
GetDisplayIndex	Returns the 0 based index of which VGA device is currently being displayed in the GUI. Only useful if more than one VGA device is active within a BSD file.
SetDisplayIndex <n>	Sets the 0 based index of which VGA devices output is to be displayed in the GUI. Only useful if more than one VGA device is active within a BSD file.
Wait	Provides a "*WAIT UNTIL STOPPED*" feature.
NGo	Provides a non-blocking "*GO*" command.
DisplayScreenShot <index> <filename> <format>	*"DisplayScreenShot"* takes a screen shot. This command supports multiple displays *Index* is a number that identifies the desired display. An *Index* of 0 means that a screen shot from display 0 will be taken. *Filename* is the name of the snapshot file. The file name includes the full pathname for the file, any valid path drive names ('C:') or server names (*servername*\) can be used. If a pathname is not given the current default path is used. *Format* must be one of the formats that *GetScreenShotFormats* returns (e.g., BMP or PNG).
GetScreenShotFormats	This command gives the list of supported formats that can be used.
LogConsoleStdErr	"*LogConsoleStdErr*" reports if *stderr* logging is currently enabled.
SetLogConsoleStdErr <0 | 1>	"*SetLogConsoleStderr*" cause console logging to go to *stderr* (1) or *stdout* (0). The default is the current behavior of logging to *stderr*.
ForceSingleStep <0 | 1>	Enabled (1) or disables (0) single stepping.
XTRInstDmpFile <FileName>	Dumps instruction to file <FileName>.
LogIO <device> | <all> <feature> | reset <0 | 1>	Enables (1) or disables (0) IO logging <feature> for <device> or <all> devices. Supported IO logging features are: PCI, IO, IOfpdis, MEM, MEMfpdis and GETMEMPTR. The *reset* options sets the selected <feature> on <device> or <all> devices to its default value.

Automation Command	Description
GetLogIO <device>	Returns IO logging status of <device>. For example, *GetLogIO "USB Jumpdrive"* returns the following information: PCI: Disabled IO: Disabled IOfpdis: Enabled MEM: Disabled MEMfpdis: Enabled GETMEMPTR: Disabled
Fastpath <device> | <all> <i | m>	Enables the IO <i> or MEM <m> fastpath for the given <device> or <all> devices.
GetFastpath <device> | all <i | m>	Returns *enabled* or *disabled* depending on if fastpath is enabled or disabled for the given <device> or *all* devices. The <i> option returns the IO fastpath status. The <m> option returns the MEM fastpath status.
SetVGAQuantum <time>	Sets the quantum value for the VGA signature mechanism. If the VGA signature matches with any of the preset golden VGA signatures the simulation stops.
GetVGAQuantum	Returns the quantum value for the VGA signature mechanism.
GenerateVGASignature <index>	Returns the VGA signature for the present screenshot. It is an MD5 sum generated from the contes of the present screen.
SetGoldenVGASignature <index>	Sets golden signature(s) needed for comparision by the VGA signature mechanism.
EnableVGASignature <0 | 1>	Enables (1) or disables (0) the VGA signature mechansim.
SetSyncQuantum <time (nanoseconds)>	Applies the MP Quantum <time> across all machines (see also *SetMPQuantum*).
GetSyncQuantum	Returns the MP Quantum value in nanoseconds set via *SetSyncQuantum* (see also *GetMPQuantum*).

A.7.2 IDE

```
1 simnow> ide.usage
```

Automation Command	Description
Image {master|slave|0|1} <filename>	Creates a volume for the given disk image (For e.g. '*ide.image 0 i:\c0d0.img*').
GetImage {master|slave|0|1}	Displays the disk image for the given volume.
Journal {master|slave|0|1} {off|on|0|1}	Turns journaling *on* or *off* for specified drive. For instance, '*ide.journal master on*' turns on journaling for master drive.

Automation Command	Description
JournalStatus {master|slave|0|1}	Returns *enabled* or *disabled* if journaling is enabled or disabled for specified drive.
JournalSize {master|slave|0|1}	Returns the journal size for specified dirve.
JournalSave {master|slave|0|1} <filename>	Saves the contents of the primary or slave disk journal to a file.
JournalLoad {master|slave|0|1} <filename>	Loads the contents of the primary of slave disk journal from a file.
JournalCommit {master|slave|0|1}	Commits the contents of the disk journal on the master or slave drive to the disk image that drive represents.
JournalClear {master|slave|0|1}	Clears the journal - discards any changes made to the drive.
JournalDebug {master|slave|0|1}	This may no longer do anything - it originally enabled a debug verification mode.
DVDROMStatus {master|slave|0|1}	Displays the status for the DVD-ROM device or a particular volume.
SetDVDROM {master|slave|0|1} {off|on|0|1}	Sets master or slave to DVD-ROM device.
Eject {master|slave|0|1} {off|<filename>}	This command is valid only for drives configured as ATAPI. The command will set the "*Media Ejected*" flag to true, and will optionally set a new image file to [File]. Us the special name "*off*" (without the quotes) if you want to leave the drive without an image file (i.e. empty) after the eject.
DMADelay {master|slave|0|1} <usec delay>	Sets the DMA delay for specified drive (master or slave) to '*usec delay*'.
Noise {off|on|0|1}	Turn *on* to print debug messages.
SetImageType {master|slave|0|1} {ID, RAW, AUTO}	This command is used to tell SimNow which type of hard disk image is used. ID indicates that the hard disk image contains an ID block. RAW indicates that the hard disk image is a sector-by-sector copy (identical to the source). AUTO indicates that SimNow will try to identify the used type of hard disk image automatically.
GetImageType {master|slave|0|1}	Returns the current image type setting, ID, RAW or AUTO. See SetImageType.

A.7.3 USB

```
1 simnow> usb.usage
```

Automation Command	Description
log (enable|disable) {mifsopt}	Enables or disables Memory (m), Interrupt (i), Frame (f), StateChange (s), PCI Config (p), Transfer (t), or/and IO (o) logging.

A.7.4 CMOS

```
1 simnow> cmos.usage
```

Automation Command	Description
Load <filepath>	Loads CMOS data stored at 'filepath'. For example '*cmos.load c:\cmos.dat*'.
Save <filepath>	Saves CMOS data to 'filepath', e.g. '*cmos.save c:\cmos.dat*'
SetTime <seconds> <minutes> <hours> <days since Sunday> <day of the month> <months since January> <years since 1900>	Sets CMOS Time to specified time. For instance 'cmos.*SetTime 00 00 12 00 31 12 14'* sets the CMOS time to Sunday December 31th, 2004, at 12:00:00.
GetByte <addr>	Returns byte in CMOS that is stored at address '*addr*'.
SetByte <addr> <data>	Sets byte in CMOS at address '*addr*' to value stored in '*data*'.
GetData	Dumps complete CMOS.
GetRamSize	Returns the CMOS RAM size in bytes.
ClearTo <value>	Sets entire CMOS to specified value '*value*'.

A.7.5 ACPI

```
1 simnow> acpi.usage
```

Automation Command	Description
PowerButton	Triggers *PowerButton* ACPI message.
SleepButton	Triggers *SleepButton* ACPI message.

A.7.6 Floppy

```
1 simnow> floppy.usage
```

Automation Command	Description
SetFloppy <A/B(0|1)> <filename>	Assigns a floppy image file '*filename'* to drive '*A*' or '*B*'.
GetFloppy <A/B(0|1)>	Returns the assigned floppy image file of drive '*A*' or '*B*'
EjectFloppy <A/B(0|1)>	The command will set the "*Media Ejected*" flag of drive '*A*' or '*B*'.

A.7.7 Debug

```
1 simnow> debug.usage
```

Automation Command	Description
Enable	Enables the Debugger and opens a debug dialog window, if GUI is enabled.

Disable	Disables the Debugger and closes debug dialog window, if GUI is enabled.
Attach <Processor Num>	Attaches debugger to specified processor.
ExecCmd <Command>	Executes the debug command specified in '*command'*, see Section 10.2, "*Debugger Command Reference*", on page 151.
MemDump	Dumps 128-bytes of memory.
DisDump	Dumps disassembly.
RegDump	Dumps all CPU registers.
MsgDump	Dumps debug messages.
WhichProc	Returns the processor number which the debugger is currently attached to.
EnableStatus	Returns '*enabled*' if debugger is enabled, '*disabled*' if debugger is disabled.
GetConfig	Displays the current configuration.

A.7.8 AMD-8151™ AGP Bridge

```
1 simnow> amd8151.usage
```

Automation Command	Description
SetRev <Rev>	Sets the internal Chip revision number of the AMD-8151 AGP device, value must be between 1 and 255.
GetRev	Gets the internal Chip revision number of the AMD-8151 AGP device.

A.7.9 VGA

```
1 simnow> vga.usage
```

Automation Command	Description
Bios <filename>	Loads the specified BIOS file.
GetBios	Returns the active BIOS file name.
VGA (0|1)	1 enables the VGA, 0 disables it.
GetVGA	Returns current status of the VGA registers, *true* if enabled and *false* if disabled.
GetConfig	Displays VGA configuration.

A.7.10 Serial

```
1 simnow> serial.usage
```

Previous versions of the simulator always used only the named-pipe format. Because of this, the named-pipe was created as soon as the BSD was loaded. Because the new version allows you to dynamically alter the communications method, the transport is not created until you hit "go" for the first time (or after making any change to the transport method). What this means is that if you are using a named-pipe, you will have to press "go" before the named-pipe is actually created

Automation Command	Description
SetLoopback (0|1)	0 disables loop back, 1 enables loop back.
GetLoopback	Returns '*true*' if loop back is enabled; otherwise it returns '*false*'.
GetCommPort[1]	Returns information regarding how the simulated serial port is configured. The result will be either: • `\\.\pipe\SimNow.COMn` This indicates that data is being transported through a named-pipe with the given name. The "n" will be either 1 for the first serial port, or 2 for the second serial port. • `\\.\COMn 57600` This indicates that data is being transported through the given serial port on the host machine using a baud rate of 57600. • `none` This indicates that data written to the simulated serial port is discarded, and no data is ever received.

[1] This only applies to the Windows® version of the simulator and not to the Linux version.

Automation Command	Description
SetCommPort[1] <none | pipe | COMn BAUD>	Sets the mode of communication you want to use with the simulated serial port. • `pipe` Tells the simulator to use a named-pipe as the method of transport for serial data to/from the simulated machine. The pipe name will be of the form "\\.\pipe\SimNow.COMn", where "n" will be 1 for serial port 1 and 2 for serial port 2. The name is not user configurable. • `COMn` Tells the simulator to use one of the host serial ports (identified by "n") as the transport for data to and from the simulated machine. "n" can be any value between 1 and 255, and must be an actual COM port that is present on the host system. Regardless of the configuration of the simulated COM port, the host COM ports baud rate is configured depending on the BAUD parameter, with 8 bit data, no parity, 1 stop bit. "BAUD" can be one of the following values (1200, 2400, 4800, 9600, 14400, 38400, 56000, 57600 or 115200). See also Section 11.1, "*Kernel Debugger*", on page 155. • `none` Tells the simulator to discard any written data, and always return "receiver empty" on reads.
SetMultiplier nMultiplier	Use the SetMultiplier automation command to specify the baud rate delay time used to make the serial based communication to Microsoft's kernel debugger in some cases much more stable. A valid nMultiplier value must be in the range of "nMultiplier>=1 and nMultiplier<=100". For example to delay the baud rate by 1/00th of normal you would enter "SetMultiplier 1". The default for nMultiplier is 100.
GetMultiplier	Returns the current value of "nMultiplier".

A.7.11 HyperTransport™ Technology Configuration

```
1 simnow> sledgeldt.usage
```

Automation Command	Description
Link (0|1|2) (0|1)	Enables or disables link 0, 1 or 2. For example '*sledgeldt.link 0 1*' enables link 0 and '*sledgeldt.link 0 0*' disables link0.
LinkStatus (0|1|2)	Returns the link status of link 0, 1 or 2.
LinkWidth (0|1|2) (8|16)	Sets link width to 8 or 16 bit of link 0, 1 or 2.
GetLinkWidth (0|1|2)	Returns link width in bits of link 0, 1 or 2.
GetConfig	Displays LDT configuration.
LogDMA (0|1)	Enables (1) or disables (0) DMA logging.

DMALogStatus	Returns '*enabled*' if logging is enabled otherwise it returns '*disabled*'.

A.7.12 8th Generation Northbridge

```
1 simnow> sledgenb.usage
```

Automation Command	Description
LogHT (0|1)	Enables (1) or disables (0) logging.
HTLogStatus	Returns '*enabled*' if logging is enabled otherwise it returns '*disabled*'.
LogPCIConfig (0|1)	Enables (1) or disables (0) PCI Config logging.
PCILogStatus	Returns '*enabled*' if PCI Config logging is enabled otherwise it returns '*disabled*'.
GetConfig	Displays Northbridge logging configuration.
ProductFile <FileName>	Loads the specified product file "*FileName*".

A.7.13 DBC

```
1 simnow> dbc.usage
```

Automation Command	Description
GetParam	Returns disk block cache parameters (size, depth and bits).
SetParam <size> <depth> <bits>	Sets disk block cache parameters.

A.7.14 AMD-8111™ Device

```
1 simnow> 8111.usage
```

Automation Command	Description
BaseID (00|01)	This specifies the HyperTransport™ protocol base unit ID. The IC's logic uses this value to determine the unit IDs for HyperTransport request and response packets. The Base ID must be 00 or 01.
GetBaseID	Returns the HyperTransport base unit ID (BUID).
HtInterrupts (0|1)	Enables (1) or disables (0) HyperTransport interrupts.
HtIntStatus	Returns '*enabled*' if HyperTransport interrupts are enabled; otherwise it returns '*disabled*'.
IoLog (0|1)	Enables (1) or disables (0) IO logging.
IoLogStatus	Returns '*enabled*' if IO Logging is enabled; otherwise it returns '*disabled*'.
MemLog (0|1)	Enables (1) or disables (0) IO logging.
MemLogStatus	Returns '*enabled*' if Memory Logging is enabled; otherwise it returns '*disabled*'.
SmiSciLog (0|1)	Enables (1) or disables (0) IO logging.
SmiSciLogStatus	Returns '*enabled*' if SMI SCI Logging is enabled; otherwise it returns '*disabled*'.

GetConfig	Displays the current AMD-8111 configuration.

A.7.15 EHC

```
1 simnow> ehc.usage
```

Automation Command	Description
log (enable | disable) {mp}	Enables or disables Memory (m) and PCI Configuration (p) logging.

A.7.16 Journal

```
1 simnow> journal.usage
```

Automation Command	Description
GetParam	Returns 'Super Block Size', 'Index Block Size', 'Index Levels', 'Disk Block Size' and 'Maximum Disk Size'.
SetParam <Super Block Size> <Index Block Size> <Index Levels> [<Disk Block Size>]	Sets journal parameters.

A.7.17 CPU

```
1 simnow> cpu.usage
```

Automation Command	Description
LoadAnalyzer <analyzer_file> [<args>]	Loads the analyzer '*analyzer_file*' with specified arguments '*args*').
ShowAnalyzers	Shows all loaded analyzers.
EnableAnalyzer <num> <0|1>	Enables (1) or disables (0) analyzer specified by '*num*'.
UnloadAnalyzer <num>	Unloads analyzer specified by '*num*'.
MCAFault <bank> <GenerateMCAFault(0|1)> <Status Reg> <Address Reg>	Causes a generic MCA fault if *GenerateMCAFault* is *true* (1) at specified Bank, *AddressReg* and status.
ProductFile <FileName>	Use product file to set fuses and configure CPU and Northbridge.
CodeGen <command> <args>	Sets or disables and enables code generator settings and options. *Command* must be one of the commands shown in Table 15-13. *Args* depends on the *command* parameter, see Table 15-13.
DumpProfile [<blocks-to-dump>]	This command is limited to showing a profile of blocks, without symbols, based on the current epoch. For more information please refer to Section A.7.17.1, "*Profiling in SimNow*".

A.7.17.1 Profiling in SimNow™ Technology

Here is an example use of the profiling command and its output:

```
1 simnow> dumpprofile 3
34962861.000000 instructions executed since the last epoch
------------------------------------------------------------------
Executed 3571672 times
 CS.D=0 LongBit=0  physical_addr=00000000000e41de  eip=00000000000041de
 00000000000041de:      cmp [04f0h],aah
 00000000000041e3:      jnz $-05h
 0000000000000000:      This block's execution was 20.431234 percent of
the total since the last epoch.

------------------------------------------------------------------
Executed 229430 times
 CS.D=0 LongBit=0  physical_addr=000000000002fd99  eip=000000000000fd99
 000000000000fd99:      lodsb ds:[esi]
 000000000000fd9b:      add ah,al
 000000000000fd9d:      loop $-04h
 0000000000000020:      This block's execution was 1.968632 percent of
the total since the last epoch.

------------------------------------------------------------------
Executed 178599 times
 CS.D=0 LongBit=0  physical_addr=00000000000274b2  eip=00000000000074b2
 00000000000074b2:      mov ax,[5724h]
 00000000000074b5:      cmp ax,[371ah]
 00000000000074b9:      jbe $+61h
 0000000000000040:      This block's execution was 1.532475 percent of
the total since the last epoch.
```

The simulator contains a code profiling facility that is accessed through the *dumpprofile* automation command. There is no graphical user interface to the profiling facility at this time. Profiling in the simulator has some limitations and features not present in most systems. The limitations are that no symbolic information is present in the output and that only execution since the beginning of the last epoch (see the last paragraph for an explanation of an *epoch*) is measured. The feature which is most unusual is that the user can ask for a profile at any time, there is no profiling mechanism that needs to be enabled before execution takes place. Another feature is that all code in the system is profiled, even code executed with interrupts off, and code in all modes (16 bit mode, 32-bit legacy mode, 32-bit compatibility mode, long mode, SMM mode, etc.) is measured equally. This profiling mechanism is non-intrusive, no x86 interrupts are taken and profiling does not affect the target machine's selection of code paths at all.

The *dumpprofile* command by itself causes all profile blocks to be displayed. This output can be quite voluminous. The user can select just the most frequently executing blocks by using an optional numeric argument. For example, "*dumpprofile 10*" will dump the ten most frequently executing blocks. Blocks are ordered by their frequency of execution, not weighted by the number of instructions in a block. Therefore, a short block executing 100 times will be displayed before a long block executing 99 times. In this example, the short block represents fewer total instructions executed. The sense of time that the simulator uses is quite simple, each instruction takes one "instruction count", with REP instructions taking one extra count per iteration. Therefore, profiles from the simulator can differ substantially from those obtained from other tools.

The simulator works by translating guest x86 instructions to long-mode user-mode instructions which it then executes. These translated instructions are grouped into blocks called *translations*. These translations exist in a translation buffer, which is typically about 64 MB. When the translation buffer is full and space for another translation is needed, the simulator disposes of the contents of the translation buffer and starts a new epoch. An epoch, in SimNow terms, is the period of execution between the flushing of the translation cache. It is only the period from the start of the current epoch to the issuance of the *dumpprofile* command that the profile will cover.

A.7.17.2 CPU Code Generator Commands

Table 15-13 describes all available Code Generator commands and their arguments.

command	args	Description
Help	None	Displays an overview of all available commands.
param	None	Displays the current state of the configurable code generator parameters.
param	parameter	Displays the current value of <*parameter*>, e.g., "cpu.codegen param FastFloat".
param	parameter value	Sets the current value of <*parameter*> to <*value*>. For example, "cpu.codegen param FastFloat 0" disables "FastFloat".
enable	Boolean Parameter	Changes the current value of one boolean parameter to true. For example, "cpu.codegen enable FastFloat" enables "FastFloat".
disable	Boolean Parameter	Changes the current value of one boolean parameter to false. For example, "cpu.codegen disable FastFloat" disables "FastFloat".
optimize	accuracy	Changes several parameters to the conservative setting.
optimize	speed	Changes several parameters to the default aggressive setting.

Table 15-13: CodeGen Command Overview

A.7.18 Emerald Graphics

```
1 simnow> emerald.usage
```

Automation Command	Description
FrameBufSize <size>	*FrameBufSize* sets the size of the frame buffer in Megabytes. The size must be a power of 2. The value placed in this option is only read at reset. The frame buffer size can not be dynamically modified.
FrameBufGetSize	Returns the size of the frame buffer in Megabytes.
Accel (0|1)	Enables (1) or disables (0) the Accelerator used by the Video driver.
GetAccel	Returns true if Accelerator is enabled; otherwise it returns false.
VBE (0|1)	Enables (1) or disables (0) VESA BIOS Extensions.

Automation Command	Description
GetVBE	Returns true if VESA BIOS Extensions is enabled; otherwise it returns false.

A.7.19 Matrox MGA-G400 Graphics

```
1 simnow> mgag400.usage
```

Automation Command	Description
SetTexmap (0 | 1)	Enables (1) or disables (0) the texture units. By default the texture units are disabled.
SetCardType CARDID	Sets the MGA-G400 type to *CARDID*. Valid values for *CARDID* are: 6648, 888, 6616, and 824.
GetCardType	Returns the current *CARDID* value.

A.7.20 PCI Bus

```
1 simnow> pcibus.usage
```

Automation Command	Description
DeviceID <SlotID> <DeviceID>	Sets the DeviceID to '*DeviceID*' on slot '*SlotID*'.
GetDeviceID <SlotID>	Returns the DeviceID of specified slot '*SlotID*'.
BaseIRQ <SlotID> (a|b|c|d)	Sets the Base IRQ of slot '*SlotID*' to A, B, C or D.
GetBaseIRQ <SlotID>	Returns the Base IRQ of slot '*SlotID*'.
Slot <SlotID> (0|1)	Enables (1) or disables (0) slot wit specified '*SlotID*'.
SlotStatus <SlotID>	Returns enabled if slot '*SlotID*' is enabled, otherwise it returns disabled.
GetConfig	Displays PCI Bus configuration information.

A.7.21 SIO

```
1 simnow> sio.usage
```

Automation Command	Description
BreakOnLock (0|1)	The Lock (1) or Unlock (0) Registers option activates the breakpoint anytime the lock or unlock sequence is hit.
GetLockStatus	Returns enabled if *BreakOnLock* is enabled; otherwise it returns disabled.
BreakOnRead (0|1)	Enable (1) or disable (0) breakpoints whenever any of the device configuration registers is read.
GetReadStatus	Returns enabled if *BreakOnRead* is enabled; otherwise it returns disabled.
BreakOnWrite (0|1)	Enable (1) or disable (0) breakpoints whenever any of the device configuration registers is modified.
GetWriteStatus	Returns enabled if *BreakOnWrite* is enabled; otherwise it returns disabled.
GetConfig	Displays SIO configuration information.

A.7.22 Memory Device

```
1 simnow> memdevice.usage
```

Automation Command	Description
Save <filename>	Creates file '*filename'* and saves the contents of the currently loaded ROM 'to *filename*'.
Load <filename>	Loads the specified MemDevice '*filename'* to defined address '*BaseAddress'*.
BaseAddress <value>	*'Value'* is the base address of the device in hex.
GetBaseAddress	Returns the base address of the device in hex.
SizeInBlocks <value>	*'Value'* is the total size of the memory device, given in decimal value for the number of 32-Kbyte blocks (32-Kbyte blocks are used because not initialized memory is dynamically allocated when addressed in 32-Kbyte chunks).
GetSizeInBlocks	Returns the number of 32-Kbyte blocks allocated by this device.
InitFile <filename>	*'filename'* is the name of the binary file that is used to initialize the memory contents. Note that the device initializes memory for the content length of the file. If you specify a 512-Kbyte ROM and use a 256-Kbyte image file, the first 256 Kbytes are initialized.
GetInitFile	Returns the path and name of the init file (see above *InitFile*).
ReadOnly <0|1>	Turns (1) the memory device into a ROM. Writes to the device are ignored when the read-only option is selected.
GetReadOnly	Returns true if memory is read-only otherwise it returns false.
SystemBios <0|1>	Tells (1) the memory device that it is the system BIOS.
GetSystemBios	Returns true if memory is used as a System BIOS otherwise it returns false.
MemAddrMask <0|1>	Enables (1) or disables (0) memory-address masking. If enabled (1) it indicates that the address received by the memory device is masked by a bit mask with the same number of bits as the size of the memory device (e.g., a 256-Kbyte ROM uses an 18-bit mask, or it is masked by 0x003FFFF). This enables the ROM to be remapped dynamically into different memory address ranges in conjunction with the aforementioned chip select.
GetAddrMask	Returns true if memory-address masking is enabled otherwise it returns false.
InitValEnable <0|1>	Enables (1) or disables (0) the initialized unwritten memory option. If enabled the memory is initialized using a specified byte (see below *InitVal*) otherwise the memory is not initialized.
InitVal <hex value>	Sets byte initializer for memory that needs to be initialized.

Automation Command	Description
InitValStatus	Displays information if the initializer is used and if the memory initialization is activated.
DisableCache < 0 | 1 >	Sets memory region to cacheable (0) or non-cacheable (1).
GetCacheDisabled	Returns true if non-cacheable otherwise it returns false.
GetConfig	Displays Memory configuration information.
FlashMode < 0 | 1 >	Enables (1) or disables (0) this device to be used as a flash ROM.
FlashUpdateFile < 0 | 1 >	Enables (1) or disbales (0) writes to the flash ROM to update the ROM image.
ncHTMode < 0 | 1 >	Enables (1) or disables (0) decoding of HyperTransport messages.
ForceInitFile <filename>	The *ForceInitFile* command allows the user to change the BIOS ROM path once the simulation has already started. This is legitimate only when the new BIOS ROM is a byte-for-byte copy of the initial BIOS ROM that simulation began with (i.e., same file, different path).
GetCommandSequence	Prints which of the two command sequences the flash device is programmed to.
CommandSequence < 0 | 1 >	0-SST, 1-ATMEL. Allows to set the command sequence to SST or ATMEL.
GetFlashMode	Tells you if the device is configured to act as a flash memory.
FlashMode < 0 | 1 >	Allows the user to set the memory device as flash memory.

A.7.23 Raid

```
1 simnow> raid.usage
```

Automation Command	Description
Noise [{enable|disable}]	Enable to print debug messages; otherwise disable.
RomImage <File name>	Allows a boot ROM image to be supported - at the moment the emulation does not work with any known ROM images.
SetVolume <Vol #> <Image file> [<Journal file>]	This was the original way to setup the image and journal files - rather than having two separate commands.
DeleteVolume <Vol #>	Undoes the Image or Journal commands and puts the volume back in an unintialized state.
Sync	This command flushes the in-memory caches out to the files.
Type {5304|5312}	This was supposed to allow support for both the 5304 (default) and 5312 cards - the 5312 support is not well tested.
Image <Vol #> <Image file>	Creates a volume for the give disk image (For e.g., *raid.image 0 i:\c0d0.img*).
GetImage <Vol #>	Displays the disk image for the given volume.

Automation Command	Description
Journal <Vol #> {0|1}	Enables (1) or disables journaling for specified volume.
AddJournal <Vol #> [<Journal file>]	Creates a journal for the given volume number (For file-based journal: *raid.addjournal 0 i:\c0d0j1.jrn*; for in-memory journal: *raid.addjournal 0*).
ResizeJournal <Vol #> [<Old Journal> <New Journal>]	Resizes the journal for the given volume to the new journal parameters.
Commit <Vol #>	*Commit* copies back the modified data blocks from the journal to the disk image and clears the journals.
Clear <Vol #>	Clears the volume - discards any changes made to the volume.
Flatten <Vol #>	Deletes the journal added last for that particular volume.
Status [<Vol #>] [-v | -r]	Displays the status for the RAID device or a particular volume. -v option displays details regarding the statistics of performance meters implemented in the RAID device, while -r option resets the performance counters.
SetDBC <Entries> <Depth> <Block Size>	Set the parameters for disk block cache (For e.g., *raid.setdbc 32768 5 512*.
SetJournalParameters <Super Block Size> <Index Block Size> <Index Levels> <DiskBlock Size>	Set the Journal Parameters (For e.g., *raid.setjournalparameters 8192 512 3 512*).
GetJournalParameters	Displays the Journal parameters.

A.7.24 DIMM

```
1 simnow> dimm.usage
```

Automation Command	Description
PdlErrorSim (0|1)	Enables (1) or disables (0) the *PDL Error Simulation*. If enabled then the DIMM device monitors PDL settings for all RAM reads.
GetPdlErrorSim	Returns enabled if PdlErrorSim is enabled; otherwise it returns disabled.
OutOfRangeResp (0xFF | invert)	The '*Out of Range Response*' selection specifies how the data should be altered if a PDL is out of range. The *0xFF* option specifies that the return data should be forced to all ones. The Invert option specifies that the return data should be a bitwise inversion of the valid data.
GetOutOfRangeResp	Returns the specified options set by *OutOfRangeResp*.
SMBBaseAddr <addr>	The *SMB Base Address* entry selects the 8-bit address that this DIMM device responds to. The SMB address is used for the reading of DIMM SPD data.
GetSMBBase	Returns the specified SMB Base address.
ImportSPD <DimmNo> <fullpath>	*ImportSPD* provides the option of loading SPD ROM data to DimmNo from the file specified by "fullpath". The file format is an unformatted binary image, with an extension of "*.spd*".

Automation Command	Description
ExportSPD <DimmNo> <fullpath>	*ExportSPD* provides the option of saving SPD ROM data from DimmNo to the file specified by "fullpath". The file format is an unformatted binary image, with an extension of ".*spd*".
ResetPDLs <DimmNo>	*ResetPDL* sets all 16 PDL response ranges to their maximum range (0 - 255).
PDLRespRange <DimmNo> <PDLNo> <High> <Low>	Sets the PDL Response Rage of memory module 'DimmNo' and PDL 'PDLNo' to 'High' and 'Low'.
GetPDLRespRange <DimmNo> <PDLNo>	Returns the PDL response range of memory module 'DimmNo' and PDL 'PDLNo'.
GetPDLData <DimmNo>	Lists the PDL data of memory module 'DimmNo'.
GetConfig	Displays DIMM configuration details, like 'PdlRespRange', 'MBBaseAddr', 'OutOfRangeResp' and 'PdlErrorSim'.
GetMaxDimms	Returns the maximum number of DIMMs that can be simulated.
SetMaxDimms <num>	Sets the maximum number of DIMMs that can be simulated.
GetDimmDescription <DimmNo>	Returns a short description of the memory module 'DimmNo'. It displays memory type, total size, number of banks and device data width in bits.
GetDimmType <DimmNo>	Returns the DIMM type of memory module 'DimmNo'.
GetDimmSize <DimmNo>	Returns the DIMM size of memory module 'DimmNo'.
GetDimmBanks <DimmNo>	Returns the DIMM banks of memory module 'DimmNo'.
GetDimmWidth <DimmNo>	Returns the DIMM width of memory module 'DimmNo'.
GetSpdData <DimmNo>	Returns SPD data of memory module 'DimmNo'.
DeleteDimm <DimmNo>	Deletes memory module 'DimmNo' from current configuration.
GetSpdDataByte <DimmNo> <Addr>	Returns a specific SPD data byte stored at <Addr> on Dimm <DimmNo>.
SetSpdDataByte <DimmNo> <Addr> <Data>	Sets the SPD data byte <Data> at SPD-Address <Addr> on DIMM <DimmNo>.

A.7.25 Keyboard and Mouse

By default the GUI uses *keyboard.key* and *keyboard.mousemove* commands to send input to the simulator. These can be overridden using the *Gui_Key_Device* and *Gui_Mouse_Device* registry keys. For example, if you connect a USB keyboard device to the simulation, you can have keystrokes use the USB keyboard rather than the old keyboard.

```
1 simnow> keyboard.usage
```

Automation Command	Description
Key <XX> [XX...]	Forwards the specified key to the simulated system. E.g., the following command forwards the ENTER keystroke to the simulated system: *keyboard.key* 1C.

MouseMove <DeltaX> <DeltaY>	Moves the mouse cursor to relative position *DeltaX* and *DeltaY*.
MouseLeftDown	Generates a left-mouse-button-down event.
MouseRightDown	Generates a right-mouse-button-down event.
MouseLeftUp	Generates a left-mouse-button-up event.
MouseRightUp	Generates a right-mouse-button-up event.
MouseMoveAbs <X> <Y>	Moves the mouse cursor to absolute x-y position.
Log enable|disable id	Enables or disables logging.
Text	This command injects keyboard input from the command line. It takes basic text such as 'keyboard.text "dir\r"'. This command can handle more complex sequences with other '\' prefixed strings (see Table 15-14).

Table 15-14 shows the currently defined prefix sequences:

Prefix	Action	Prefix	Action
\r	<RETURN>	\{f8}	<FUNCTION KEY 8>
\t	<TAB>	\{f9}	<FUNCTION KEY 9>
\\	<BACKSLASH>	\{f10}	<FUNCTION KEY 10>
\″	<DOUBLE QUOTE>	\{tab}	<TAB>
\′	<SINGLE QUOTE>	\{del}	<DELETE>
\{esc}	<ESCAPE>	\{up}	<UP ARROW>
\{f1}	<FUNCTION KEY 1>	\{down}	<DOWN ARROW>
\{f2}	<FUNCTION KEY 2>	\{left}	<LEFT ARROW>
\{f3}	<FUNCTION KEY 3>	\{right}	<RIGHT ARROW>
\{f4}	<FUNCTION KEY 4>	\{ctrl-m}	<CONTROL make>
\{f5}	<FUNCTION KEY 5>	\{ctrl-b}	<CONTROL BRAKE>
\{f6}	<FUNCTION KEY 6>	\{alt-m}	<ALT MAKE>
\{f7}	<FUNCTION KEY 7>	\{alt-b}	<ALT BRAKE>

Table 15-14: Prefix Sequences (keyboard.text)

A.7.26 JumpDrive

```
1 simnow> jumpdrive.usage
```

Automation Command	Description
LoadImage <HostFileName>	Loads the contents of the specified image file <*HostFileName*> to the memory.
SaveImage <HostFileName>	Saves the contents of the memory to an image file on the host specified by <*HostFileName*>.
ImportFile <HostFileName> <ImageFileName>	Imports the requested file into the image <*ImageFileName*> using the given host file name <*HostFileName*>.
ExportFile <ImageFileName> <HostFileName>	Exports the requested file from the image <*ImageFileName*> to the given host file name <*HostFileName*>.

Automation Command	Description
Initialize <SizeInMB>	Initialize the jump drive image with a single partition of the requested size specified by <*SizeInMB*>. The JumpDrive supports image-sizes from 64-Mbytes to 8192-Mbytes (8-Gbytes).
ImportDir <HostPathName> <ImagePathName>	Imports a directory from the host system into the jump drive. The host path name <*HostPathName*> can contain wildcards in the last element. If the last element of the <*HostPathName*> does not contain wildcards, and points to a directory, then "*" is assumed. The image path name <*ImagePathName*> must be the name of a directory. If it does not exist, it will be created.
ExportDir < ImagePathName> <HostPathName>	Exports a directory from the jump drive to the host system. The image path name <*ImagePathName*> can contain wildcards in the last element. If the last element of the <*ImagePathName*> does not contain wildcards, and points to a directory, then "*" is assumed. The host path name <*HostPathName*> must be the name of a directory. If it does not exist, it will be created.
Dir <ImagePathName>	Shows the contents of the directory path given by <*ImagePathName*>.
Free	Shows the amount of free space on the JumpDrive device.
Size <Size in MB>	This command is identical to the *Initialize* command, only it does not create a FAT32 partition on the drive. It simply sets the physical size of the device. Any formatting or initialization will still need to be done (presumably by the simulated operating system).

To initialize the JumpDrive, and copy data to it:

```
1 simnow>jumpdrive.initialize 64
```

This creates a 64-Mbyte FAT32 partition on the JumpDrive.

The following example copies the file "C:\test.bin" to the JumpDrive and places it in the "\tmp" directory. If the "\tmp" directory does not exits on the JumpDrive, it is created automatically.

```
1 simnow>jumpdrive.importfile c:\test.bin \tmp\test.bin
62.99 Mbytes Available
```

```
1 simnow>
```

This copies all files from "C:\tmp" into the root of the JumpDrive. Any subdirectories are also copied.

```
1 simnow>jumpdrive.importdir c:\tmp \
Importing c:\tmp\test.bin ---> \test1.bin
62.89 Mbytes Available
```

This example shows how to import all "*.exe" files from "C:\tmp" into the root of the JumpDrive.

```
1 simnow>jumpdrive.importdir c:\tmp\*.exe \
Importing c:\tmp\app1.exe ---> \app1.exe
Importing c:\tmp\app2.exe ---> \app2.exe
62.60 Mbytes Available
```

This example shows how to export the "app1.exe" file from the root of the JumpDrive into "C:\tmp" on the host.

```
1 simnow>jumpdrive.exportfile \app1.exe c:\tmp\
Exporting \app1.exe ---> c:\tmp\app1.exe
```

To find out what is already stored in the root of the JumpDrive device, enter the following:

```
1 simnow> jumpdrive.dir \

Directory of: \

      <DIR>       tmp
      103936      test.bin
      103936      app1.exe
      103936      app2.exe

62.60 Mbytes Available
```

To get information about how much space is left on the JumpDrive device, enter the following:

```
1 simnow>jumpdrive.free
62.60 Mbytes Available
```

To save the contents of the JumpDrive to the image file "C:\test.img" on the host's hard-disk, enter

```
1 simnow>jumpdrive.saveimage c:\test.img
```

This example shows how to load the saved JumpDrive image "C:\test.img" from the host's hard-disk into the JumpDrive

```
1 simnow>jumpdrive.loadimage c:\test.img
```

A.7.27 E1000

The NIC device provides the following automation commands that can be used to configure the device.

```
1 simnow> e1000.usage
```

Automation Command	Description
log enable|disable cmoidtr	Enables or disables message logging for PCI Config (c), MMIO (m), I/O (o), Unmasked Interrupts (i), MDI (d), Frame Transfers (t), or Frame Receptions (r).
logStatus	Displays the current log-status.
setMediatorHost [domain@]hostname[:port]	Sets the mediator connect string. The domain string and the port number are optional. The default domain string is null. The default port is 8196. The hostname is the host in which the mediator is running.
getMediatorHost	Outputs the current mediator connect string.
setMACAddress XX:XX:XX:XX:XX:XX	Sets the MAC Address to be used by the adapter.
getMACAddress	Retrieves the MAC Address being used by the adapter.
linkConnect auto|down	Restarts link negotiation (auto) for the adapter, or forces a link disconnect (down).
tune {intthrtl|rxdelay|txdelay} value	Sets certain synthetic delay- and throttle-values which gives the user the opportunity to change the default settings to get optimal results. *intthrt* sets the interrupt throttle rate to *value*. *rxdelay* sets the amount of link idle time required before generating an rx interrupt to *value*. *txdelay* sets the amount of link idle time required before generating an tx interrupt to *value*.
getTuneValues	Displays the values set by using the automation command *tune*.

A.7.28 XTR

```
1 simnow> xtrnb.usage
```

Automation Command	Description
xtrfile <filename.xml>	Sets XTR–XML file to use during playback.
debug <0|1>	Enables (1) or Disables (0) extended debug information for XTR Playback.

Automation Command	Description
xtrlogfile <filename.log>	Sets name of the log file where XTR messages should be logged. This is optional and if not used the log is directed to the simulators log.
status	Displays the status of XTR playback

```
1 simnow> xtrsvc.usage
```

Automation Command	Description
xtrenable <0|1>	Enables (1) or Disables (0) XTR Record. All other values are invalid.
xtrfile <filename.xml>	Sets the XTR-XML file for XTR Record.
XTRMemBits n	Sets number of bits for memory address bits to scan. n= 16, 32 or 48. Default is 32.
Xtrstatus	Displays the status of XTR Record.

A.7.29 ATI SB400/SB600/SB700

```
1 simnow> sb600.usage
```

Automation Command	Description
HtInterrupts (0|1)	Enables (1) or disables (0) HyperTransport interrupts.
HtIntStatus	Returns '*enabled*' if HyperTransport interrupts are enabled; otherwise it returns '*disabled*'.
IoLog (0|1)	Enables (1) or disables (0) IO logging.
IoLogStatus	Returns '*enabled*' if IO Logging is enabled; otherwise it returns '*disabled*'.
MemLog (0|1)	Enables (1) or disables (0) IO logging.
MemLogStatus	Returns '*enabled*' if Memory Logging is enabled; otherwise it returns '*disabled*'.
SmiSciLog (0|1)	Enables (1) or disables (0) IO logging.
SmiSciLogStatus	Returns '*enabled*' if SMI SCI Logging is enabled; otherwise it returns '*disabled*'.
Version	Displays the binary revision of the RD790 model.
SetPciIrqMap {BasePciIrq(0-3)} {ChipPciIrq(0-7)}	Depending on platform configuration, it maps base PCIIRQ#A/B/C/D (0-3) from PCI bridge to ATI chip internal PCIIRQ#A/B/C/D/E/F/G/H (0-7).
GetPciIrqMap {BasePciIrq(0-3)}	Returns the ATI chip internal PCIIRQ#A/B/C/D/E/F/G/H (0-7) which the specific base PCIIRQ#A/B/C/D(0-3) is mapped to.
GetPciIrqTotal	Returns the total number of chip internal PCIIRQs.

A.7.30 ATI RS480

```
1 simnow> rs780.usage
```

Automation Command	Description
SetRev <rev >	Sets the internal chip revision number of RS480 device to <*rev*>.
GetRev	Displays the internal chip revision number of the RS480 device.

A.7.31 ATI RS780

```
1 simnow> rs780.usage
```

Automation Command	Description
SetRev <rev >	Sets the internal chip revision number of RS780 device to <*rev*>.
GetRev	Displays the internal chip revision number of the RS780 device.
Version	Displays the binary revision of the RS780 model.

A.7.32 ATI RD790/RD780/RX780

```
1 simnow> rd790.usage
```

Automation Command	Description
SetRev <rev >	Sets the internal chip revision number of RD790 device to <*rev*>.
GetRev	Displays the internal chip revision number of the RD790 device.
Version	Displays the binary revision of the RD790 model.
SetPackageType <RD790 | RX780>	Sets package type to *RD790* or *RX780*.
GetPackageType	Displays current package type.

A.7.33 ATI RD890S/RD890/RD780S/RX880

```
1 simnow> rd890.usage
```

Automation Command	Description
SetRev <rev >	Sets the internal chip revision number to <*rev*>.
GetRev	Displays the internal chip revision number.
Version	Displays the binary revision.
SetPackageType <RD890S | RD890 | RD870S | RX880>	Sets package type to *RD890S, RD890, RD870S, or RX880*.
GetPackageType	Displays current package type.

Index

***

*.ROM....179
*.SPD....180

A

A20....78
ACPI....8
Address-Translation Cache....30
AGP....65, 69
AMD 3DNow!™ Technology....220
AMD 8th Generation Integrated Northbridge ..86
AMD-8111™ Device....90
AMD-8132™ PCI-X® Controller....14, 99
AMD-8151™ Device....102
AT24C Device....14, 107

B

Base Address....167
Baud Rate....80
BIOS ROM....167
BSD file....40, 49

C

Checkpoint....49
Chip-Select....82
Clearing CMOS....169
CMOS....91, 169
Code Generator....238
Code Pages....184
COM1....78
COM2....78
Commit....43, 92
Configuration File....5
Console Window....29
CPUID....182
CR4.PCE....184
Create Device Connection....11
Creating Floppy-Disk Image....171
Cycle-Accurate....1

D

Debug
- Find Pattern....151
- Read/Write MSRs....150
- Reading CPU MSRs....150
- Reading PCI Configuration Registers....150
- Set Breakpoint....147
- Single-Stepping....148
- Skip Instruction....149
- Stepping Over....148
- View Memory....149

Deprecated Devices....10
Device ID....97
Device List....10
Devices Window.... 9
DHCP....126
Diagnostic Ports....29
DIMM....59
Disable USB Port....90
Disk Journaling....43, 92
DiskTool....161
Double Fault....184
DVD-/CD-ROM....35

E

ECC....64
EOT....112
Error Log....143
EXDI....108

F

Fan....78
FAQ....173
Flash-ROM....83
FLDENV....184
Floppy-Disk....44
Frame-Buffer....66
FRSTOR....184
FSAVE....184
FSTENV....184

G

Gateways....126
GDB....156
GPIO....78
Graphics....2, 65, 69

H

Host Operating Systems.... 3
HyperTransport™ Technology
- Coherent....86
- Link....88
- Link-capable devices....86
- Messages....87
- Non-Coherent....86
- Tunnel....14, 99
- Upstream Link....100

I

Insert CD-ROM....43
INT/IOAPIC....90
IR 78
IRQ-Routing Pin....96

J

Journaling 92, 93
Journals....104
Joystick....78

K

Kernel Debugger 108, 155

L

Linux
  Loopback Device 44
Log
  CPU Cycles....170
  Dissassembly 56
  Exceptions 56
  I/O Logging 144
  I/O Read/Writes....56
  Linear Memory Accesses 56
  Register State Changes 56
LPC/ISA Bridge 90
LPT1....78

M

MAC Address....127
Mediator Daemon....126
Memory Configurator....167
Message Log....141
Microcode Patching....184
Microsoft DirectX 9 2
MIDI....78
MIPS....30
Modify PCI Configuration Space 140
Mouse Cursor 173
Multiple Virtual-Mappings....184

N

Named-Pipe 155, 156, 157, 158, 232, 233, 234
Nested-Task....219

P

Pacifica Virtualization Technology 150
Partition 163
PCI-X 98
PCI-X Configuration Cycle 101
PDL 59
  Enable Error Simulation 62
  Error Simulation Control 62
  Reset 64
Performance-Monitoring Counter 219
Physical Drives....162
Play....7
PnP Monitor 130
  DDC....130
  VESA....130
POST....29
Prescott New Instruction 221
PS/2 mouse 78

R

RAID 104
RAM
  Memory Device 81
  Size 168
RDPMC 184
Reset 7, 45
ROM....81

S

Scripting 224
SEGV....4, 31
SEM....183
Shell....224
SimStats....29
Single-Stepping .... *See Debug*
Slowdown 1
SMB....14, 84, 85
  Hub 14, 105
SMB Base Address....62
Soft Power 8
Soft Sleep.... 8
Solo.bsd 49
SPD....168
  Export 64
  Import 64
SPD Data 168
Stepping Over .... *See Debug*
Stop.... 7
Stop XTR....111
Super IO....78
SVGA 65, 69
Switching CD Images 175
System Requirements 3

T

TCache....30
TLB 57
Triple Fault 184
TSS 219

U

Usage Command....225
User Defined Keys.... 5

V

VGA 65, 69
Virtual-Address Space 4

W

Winbond W83627HF 78
Workspace 10

X

XTR.....110
Playback.....111
Recording.....111
Stop Recording 111
XVGA 65, 69